FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Performance of Software for the Full Symmetric Eigenproblem on CRAY T3E and T90 Systems

Inge Gutheil, Ruth Zimmermann

FZJ-ZAM-IB-2000-07

Juli 2000

(letzte Änderung: 04.07.2000)

Performance of Software for the Full Symmetric Eigenproblem on CRAY T3E and T90 Systems

I. Gutheil and R. Zimmermann

John von Neumann-Institut für Computing
Zentralinstitut für Angewandte Mathematik
Forschungszentrum Jülich GmbH
July 4, 2000

Abstract

MPP systems with a message-passing programming paradigm such as CRAY T3E still play an important role in high performance scientific computing and so there is need for basic numerical software for those systems. There are several libraries for numerical linear algebra computations with message-passing available most of which are public domain libraries. The eigensystem solvers of some of these libraries are compared taking into account not only performance but also user-friendliness. Furthermore, the results attained on a PVP CRAY T90 vector supercomputer with shared-memory parallelism are discussed.

Contents

1 Introduction

Although shared-memory or virtually-shared-memory architectures will more and more dominate scientific computing the message-passing paradigm will still play an important role in very large scale computations where a couple of SMP systems can be clustered to solve problems which are too large for a single SMP machine. So there still will be need for message-passing libraries for basic numerical computations such as the solution of the symmetric eigenvalue problem. Performance and user-friendliness are important factors for acceptance of those libraries even if the routines will only be used as building blocks to develop packages for the solution of specific problems. The computation of the eigenvalues and eigenvectors of a full symmetric matrix is a time-consuming subtask in many MPP application codes, especially in the field of quantum chemistry.
On the other hand shared-memory parallel routines like those for the PVP system CRAY T90 may serve as building blocks for computations on clusters of SMP machines where they can be used on a single SMP machine in combination with message-passing algorithms among many SMP machines.
In this study we will not only measure performance of different library routines for the solution of the symmetric eigenvalue problem but also give the potential user some advice on the flexibility of the routines, the possibility to influence performance, scalability and efficiency of the routines and the user-friendliness of the different libraries. We think it is an important factor in code development how long it takes to get used to a computing model.

2 Libraries and Routines Examined

For the massively parallel system CRAY T3E apart from the public domain libraries *ScaLAPACK* [1] (which is partly integrated in the *Cray Scientific Libraries 3.0* (*libsci*) [2]) and *Global Arrays* [3] we also studied the commercially available *NAG Parallel Library* [4] which contains an interface to *ScaLAPACK* to make it easier to use. *ScaLAPACK*, *Global Arrays* and *NAG Parallel Library* are all Fortran 77 libraries, *ScaLAPACK* and *Global Arrays* may be called from C, too. For comparison, single node programs from *libsci* and the *NAG Fortran Library* Mark 17 [5] were also studied on CRAY T3E.
On CRAY T90 we used *libsci* [2] routines and routines from the *NAG Fortran Library* Mark 17.
We concentrated on the computation of all eigenvalues and eigenvectors of a real full symmetric matrix.

2.1 Details of the Libraries Examined

The following libraries were examined:

2.1.1 Libraries for MPP Systems

- *Cray Scientific Libraries* 3.0 (*libsci*) [2]
  is contained in craylibs 3.2.0.0 from SGI/Cray.

- *ScaLAPACK* public domain version 1.6 [1]
  The complete public domain version 1.6 of *ScaLAPACK* including the sublibraries *PBLAS*, *TOOLS*, and *REDIST* together with the *BLACS*, *BlacsF77init*, and *BlacsCinit* were installed using the computing environment version 3.0 from SGI/Cray with the following options: For *ScaLAPACK*, *PBLAS*, *TOOLS* and *REDIST* the Fortran switches were "-dp -ev -e0" with default optimization, i.e. "-O2", the C switch was "-O3". For the *BLACS*, *BlacsF77init*, and *BlacsCinit* the Fortran switches were "-O3, aggress" "-ev -e0" and the C switch again "-O3". As there is a bug in the *BLACS* library from *libsci*, which does not allow the construction of subgrids, all routines using routines from *REDIST* must use the public domain version of the *BLACS* which is based on *MPI* instead of *SHMEM* as the ones from *libsci*.
  The public domain version of the *BLACS* has higher latencies for point-to-point-communication than the one from *libsci*. This results in slower performance of point-to-point-communication for small messages (sending 100 64-Bit words with Cray *BLACS* is more than twice as fast as with public domain *BLACS*), for longer messages the differences are becoming significantly smaller but are still in the range of 20 % for message sizes of 1 million words (see [17]).
  Due to different optimizations of broadcasts for *SHMEM* and *MPI* the public domain version here is faster for message sizes between 1 and 1000 but becomes again about 10-20 % slower for messages larger than 10000.
  With global combine operations the situation is the other way round: The public domain version is faster than the Cray version for messages larger than 10000 and slower for smaller messages as *MPI* ist optimized in a different way than *SHMEM*. So the overall performance of both *BLACS* libraries does not differ too much.

- *Global Arrays* 2.4
  We studied *Global Arrays* 2.4, which we installed using the makefiles delivered with the source. The computing environment was 3.1. For the solution of the symmetric eigenvalue problem *Global Arrays* uses the *PeIGS* library [6]. At the time we started our measurements there was no public domain CRAY T3E version of *PeIGS* available, but as there is a *PeIGS* library version 2 contained in *NWChem*, which is installed here, we used this in com-

bination with *Global Arrays*. The *PeIGS* library of *NWChem* was installed under UNICOS/mk 2.0.4 and the computing environment 3.2. All routines of *PeIGS* were compiled with default optimizations, i.e. "-O2" for Fortran and C.

- *NAG Parallel Library* Release 2
  NAG Parallel Library Release 2 was inspected, which we installed with *PBLAS* and *BLACS* from *libsci*. NAG promised to deliver their own *PBLAS* and *BLACS* libraries but they were not yet available and it is not clear whether they will be better optimized than those from *libsci*. The implementation of the library was produced with operating system UNICOS/mk 2.0.3, computing environment 3.0 with compiler switches "-dp -Xm -O2, aggress" for both Fortran and C, and *MPT* (Cray implementation of *MPI*) version 1.2.1.0.

2.1.2 Libraries for PVP Systems / Single Node Libraries for MPP systems

- *Cray Scientific Libraries* 3.0 (*libsci*) [2]
  is contained in craylibs 3.2.0.0 from SGI/Cray.

- *NAG Fortran Library* Mark 17 [5]
  The CRAY T3E version of the *NAG Fortran Library* was produced under UNICOS v9.1.0 with the Cray CF90 v2.0.3.0 compiler. The entire library was compiled with "-O scalar3" optimization, except for some routines not involved in our investigation.
  The BLAS are included in the *NAG Library* on CRAY T3E.

  The CRAY T90 version of the *NAG Fortran Library* was produced under UNICOS v9.1.0 with the Cray CF90 v2.0.0.2 compiler. The entire library was compiled with "-O scalar3" optimization, except for some routines not involved in our investigation.
  The BLAS are not included in the *NAG Library* on CRAY T90.

- *LAPACK* Release 2 [9]
  Since SSYEVD was'nt yet available in *libsci* on CRAY T3E, the public domain version was used for this routine and all other *LAPACK* routines not contained in *libsci* but called by other routines. The public domain version of *LAPACK* was compiled with "-O 3" optimization.

2.2 Details of the Routines Examined

The following library routines were examined:

2.2.1 MPP Routines

- *libsci* : PSSYEVX
  PSSYEVX is the expert driver routine for the solution of the real symmetric eigenvalue problem. It computes all or a selected subset of the eigenvalues and (optionally) the corresponding eigenvectors. The computation is done in three steps: (see [10])

  1. reduction of the dense matrix to tridiagonal form using Householder transformations;
  2. computation of the eigenvalues of the tridiagonal matrix using bisection and computation of the eigenvectors using inverse iteration;
  3. back-transformation of the eigenvectors;

  Like the corresponding *LAPACK* routine SSYEVX PSSYEVX tries to reorthogonalize eigenvectors belonging to clustered eigenvalues. As this is a serial bottleneck for large clusters and has to be done on a single node for one cluster, *ScaLAPACK* has to deal with that problem (see [1]). There is an additional variable ORFAC where the user can specify which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within tol=ORFAC*norm(A) of each other are to be reorthogonalized. However, if the workspace is insufficient, tol may be decreased until all eigenvectors to be reorthognalized can be stored in one process. So the user can also influence reorthogonalization by providing more or less workspace. No reorthogonalization will be done if ORFAC equals zero. A default value of 10^{-3} is used if ORFAC is negative. The measurements were done with this default value unless mentioned otherwise.
  ScaLAPACK is written with blocked algorithms to use the highest level of BLAS routines possible. In PSSYEVX if there are no large clusters of eigenvalues most time is spent in steps 1 and 3 and these steps can be blocked to use BLAS 2 and BLAS 3 routines (see Table 2).

- *ScaLAPACK* : PSSYEV
  PSSYEV is the simple driver, which computes all the eigenvalues and (optionally) the eigenvectors of a real symmetric matrix. It is done in the same three steps as in PSSYEVX but for step 2 a modified QR algorithm developed by G. Henry[11] is performed by each processor redundantly where all eigenvalues are computed on each processor but the eigenvectors are computed in parallel. The modified QR gives rise to many calls of the BLAS 1 routine SROT (see Tables 2 and 3). PSSYEV will guarantee orthogonality and because it has O(1) communication, it will scale well though it is said to

be several times slower than PSSYEVX (see [10], p 5). This is true if no reorthogonalization of eigenvectors belonging to large clusters of eigenvalues is necessary with PSSYEVX. If eigenvalues are not clustered, PSSYEV needs about twice as many arithmetic operations as PSSYEVX (see Table 8) and less communication (see Table 6) and because of the heavy usage of BLAS 1 routines the MFLOPS for large problems are lower than the measured ones of PSSYEVX. Otherwise PSSYEV can be much faster than PSSYEVX because too much time is spent with computation on only one processor.
It was also seen, that PSSYEV was faster with one large cluster of eigenvalues than with equally spread eigenvalues, and sometimes less operations per node had to be done when the eigenvalues were clustered.
If there was a large cluster of exactly equal eigenvalues computation time decreased very much (This phenomenon is still to be investigated in detail).

- *NAG Parallel Library* : F02FQFP
  The algorithm of F02FQFP to calculate all eigenvalues and eigenvectors is based on an one-sided Jacobi method. A sequence of rotations is applied, each of which zeroes one off-diagonal entry, bringing the matrix to diagonal form. The one-sided Jacobi method needs more operations than the two-sided, but is known to be more efficient on a parallel platform, because communication is reduced. The Jacobi method shows good accuracy, but is slow, if the matrix is not strongly diagonally dominant. Therefore, the algorithm isn't competitive in the general case. Parallelism is introduced by applying several rotations simultaneously. Very good speedup values can be reached (see Table 5).

  In F02FQFP, the columns of a matrix of size n are allocated to logical processors on a 2-d grid of totally p processors row by row. Each logical processor that contains columns of the matrix contains a block of $N_b = [n/p]$ contiguous columns, except the last one, for which the number of columns held may be less than N_b. If n is not large relative to p ($n \leq [n/p](p-1)$), some processors may not contain any columns of the matrix.
  Since F02FQFP is dominated by BLAS 1 calls (see Tables 2 and 3), it cannot be efficient on CRAY T3E due to the small level 1 cache.

- *Global Arrays* : GA_DIAG_STD
  GA_DIAG_STD calls PDSPEV from the *PeIGS* library. All eigenvalues and eigenvectors are computed. It uses the same three computational steps as PSSYEVX but with a modification in step 2, where a subspace inverse iteration and reorthogonalization scheme for finding basis vectors for degenerate eigen-subspaces is used. In particular, inverse iteration to convergence is performed on all of the eigenpairs in a cluster in a perfectly parallel fashion. Orthogonalization is then performed in parallel using all the processors

storing the unconverged eigenvectors of the cluster. Exactly two iterations of "inverse iteration followed by orthogonalization" are always done. The key here is that two iterations of modified Gram-Schmidt are used in the first orthogonalization. This algorithm usually yields highly orthogonal eigenvectors, but it is not always guaranteed since a convergence test is not currently done (see [6]).
The implementation of PDSPEV is done in C using BLAS 1 routines for all computationally intensive parts and BLAS 2 and BLAS 3 routines are never called (see Tables 2 and 3). So this routine also cannot be very efficient on CRAY T3E.

2.2.2 PVP Routines and Single Node Routines on MPP Systems

- *libsci* : SSYEV
  SSYEV is part of the public domain package *LAPACK*. After scaling the real symmetric matrix is reduced to tridiagonal form by an orthogonal similarity transformation via the routine SSYTRD. If only eigenvalues are required, SSTERF is called to compute all eigenvalues of a symmetric tridiagonal matrix using the Pal-Walker-Kahan variant of the QL or QR algorithm. If also eigenvectors are desired, SORGTR is called to generate a real orthogonal matrix Q which is defined as the product of $n-1$ elementary reflectors of order n, as returned by SSYTRD. Then SSTEQR computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method.

- *libsci* : SSYEVX
  In this *LAPACK* routine, if the user chooses the parameter ABSTOL to be less or equal zero, and if all eigenvalues are desired, SSYEV is called. Otherwise, after scaling and reduction to tridiagonal form, SSTEBZ calculates selected eigenvalues by bisection and STEIN determines selected eigenvectors by inverse iteration.

- *libsci* (CRAY T90), *LAPACK* (CRAY T3E) : SSYEVD
  If only eigenvalues are required, the calculations are the same as for SSYEV. Otherwise SSTEDC computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method. The code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the CRAY X-MP, CRAY Y-MP, CRAY C-90, or CRAY-2. It could conceivably fail on hexadecimal or decimal machines without guard digits. The algorithm shows very good performance, but is very memory-consuming: more than four times the

square of the matrix size is needed.

- *NAG Fortran Library* : F02FAE
  F02FAE is derived from the *LAPACK* routine SSYEV.
  Using BLAS from *libsci* the results are the same as for SSYEV. Using BLAS from *NAG Fortran Library* on CRAY T3E the proportion of BLAS 3 becomes larger but also execution times get higher due to the non-optimized BLAS 3 routine SGEMM.

2.3 User Interfaces of the MPP-Libraries

ScaLAPACK and *NAG Parallel Library* are classical Fortran 77 subroutine libraries whereas *Global Arrays* supports a global view of distributed data objects and supplies the user with routines to fill those objects with data and manipulate them.

2.3.1 ScaLAPACK

ScaLAPACK as a parallel successor of *LAPACK* [9] attempts to leave the calling sequence of the subroutines as much as possible unchanged in comparison to the respective sequential subroutines from *LAPACK*. Therefore, *ScaLAPACK* uses so-called descriptors, which are integer arrays containing all necessary information about the distribution of a matrix. This descriptor appears in the calling sequence of the parallel routine instead of the leading dimension of the matrix in the sequential one. For example the sequential driver routine SSYEVX for the solution of the full real symmetric eigenvalue problem has the following calling sequence

```
CALL SSYEVX(JOBZ,RANGE,UPLO,N,A(1,1),LDA,VL,VU, &
           IL,IU,ABSTOL,M,W,Z(1,1),LDZ,WORK,    &
           LWORK,IWORK,IFAIL,INFO)
```

whereas the *ScaLAPACK* routine PSSYEVX is called

```
! Call of PSSYEVX with descriptors and the global
! starting indices of the submatrix required
CALL PSSYEVX(JOBZ,RANGE,UPLO,N,A,1,1,DESCA,VL,VU,   &
           IL,IU,ABSTOL,M,NZ,W,ORFAC,Z,1,1,DESCZ,WORK, &
           LWORK,IWORK,LIWORK,ICLUSTR,GAP,INFO)
```

The additional variable NZ tells how many eigenvectors actually converged. LIWORK indicates the size of IWORK, which depends on the number of processors as well as on the matrix size. All other additional variables deal with the problem of reorthogonalization of eigenvectors belonging to clustered eigenvalues. With ORFAC the user can set the threshold for orthogonalization (see section 2.2). The array ICLUSTR contains indices of eigenvectors belonging to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace. The array GAP contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array ICLUSTR.

The user has to take care of the data distribution. He only gets some auxiliary routines in a *TOOLS* sublibrary to help in converting global to local indices and vice versa. It is completely left to the user to put the right local part of the matrix to the right places and to put the correct data to the descriptor. The users' guide and the comments at the beginning of all routines are usually sufficient to use *ScaLAPACK* correctly but for someone not used to parallel programming it is very difficult and takes a lot of time to learn how to use it.
The main steps the user has to deal with are:

```
! Create the MP * NP processor grid
CALL BLACS_GRIDINIT(ICTXT,'Row-major',MP,NP)
! Now ICTXT is the context for communication in the grid
! Find my processor coordinates MYROW and MYCOL
! NPROW should return same value as MP,
! NPCOL should return same value as NP
CALL BLACS_GRIDINFO(ICTXT, NPROW, NPCOL, MYROW, MYCOL)
! Compute local dimensions with routine NUMROC
! N is dimension of the matrix
! NB is block size
MYNUMROWS = NUMROC(N,NB,MYROW,0,NPROW)
MYNUMCOLS = NUMROC(N,NB,MYCOL,0,NPCOL)
! Allocate only the local part of A
ALLOCATE(A(MYNUMROWS,MYNUMCOLS))
! Fill the descriptors, P0 and Q0 are processor coordinates
! of the processor holding global element A(0,0)
CALL DESCINIT(DESCA,N,N,NB,NB,P0,Q0,ICTXT,MYNUMROWS,INFO)
! Fill the local part of the matrix with data
!
DO J = 1, MYNUMCOLS, NB
  DO JJ=1,MIN(NB,MYNUMCOLS-J+1)
    JLOC = J-1 + JJ
    JGLOB = (J-1)*NPCOL + MYCOL*NB + JJ
    DO I = 1, MYNUMROWS, NB
      DO II=1,MIN(NB,MYNUMROWS-I+1)
        ILOC = I-1 + II
        IGLOB = (I-1)*NPROW + MYROW*NB + II
        A(ILOC,JLOC) = function of global indices IGLOB,JGLOB
      ENDDO
    ENDDO
  ENDDO
ENDDO
...
! now computational subroutines are called
...
! undefine logical processor grid
CALL BLACS_GRIDEXIT(ICTXT)
CALL BLACS_EXIT(0)
```

2.3.2 *NAG Parallel Library*

To help with difficulties of that kind, *NAG Parallel Library* offers routines to generate and distribute matrices and vectors for use in *NAG* and *ScaLAPACK* routines, to read and write distributed data objects, and to determine for example the length of workspace needed in *ScaLAPACK* routines. This makes the *NAG Parallel Library* much easier to use for an unexperienced user. The following example shows how the data distribution used in a *ScaLAPACK* routine is done with the help of *NAG Parallel Library* routines. The most important advantage is, that the user does not

have to know, which part of the matrix has to be stored on each processor. He only has to specify a submatrix of A, i.e. A(i1: i2, j1: j2) for global indices i1, i2, j1, j2 given by a *NAG Parallel Library* subroutine.

To better understand the following code fragment, one has to know, that the error parameter `IFAIL` used in all *NAG* routines is not only an output parameter, but is also used to control the behaviour of the *NAG* programs in case of errors. If `IFAIL` is set to zero, in case of errors, the *NAG* routines print an error message and stop the execution of the user program.

```
....
!     define the MP x NP  processor grid
IFAIL = 0
NPROW=MP
NPCOL=NP
CALL Z01AAFP(ICTXT,NPROW,NPCOL,IFAIL)
!     get the actual processor coordinates MYROW, MYCOL
CALL BLACS_GRIDINFO(ICTXT,MP,NP,MYROW,MYCOL)
!     call NAG support routines to calculate the number of
!     rows  and columns  of the local part of the
!     the matrix for dimension N and blocksize NB x NB
MYNUMROWS=Z01CAFP(N,NB,MYROW,0,MP)
MYNUMCOLS=Z01CAFP(N,NB,MYCOL,0,NP)
!     descriptor for matrix A
!     processor with coordinates (0,0) has to hold
!     global element A(0,0)
CALL DESCINIT(DESCA,N,N,NB,NB,0,0,ICTXT,MYNUMROWS,INFO)
!     allocate only local part of A
ALLOCATE(A(MYNUMROWS,MYNUMCOLS))
!     Fill the local part of matrix A via user routine
!     GMATA
IFAIL=0
CALL F01ZQFP(GMATA,N,N,A,1,1,DESCA,IFAIL)
...
! now other subroutines e.g. from ScaLAPACK are called
...
!     undefine the logical processor grid
IFAIL=0
CALL Z01ABFP(ICTXT,,'N',IFAIL)
 ....

SUBROUTINE GMATA(I1,I2,J1,J2,AL,LDAL)
!     block A(I1:I2, J1:J2) of matrix A is stored in local
!     array AL; global indices I1,I2,J1,J2 are given by
!     the NAG program
REAL             :: AL(LDAL,*)
JLOC = 1
DO JGLOB = J1, J2
   ILOC = 1
   DO IGLOB = I1, I2
      AL(ILOC,JLOC)=function of global indices IGLOB,JGLOB
      ILOC = ILOC + 1
   END DO
   JLOC = JLOC + 1
END DO
RETURN
END SUBROUTINE GMATA
```

2.3.3 *Global Arrays Toolkit*

Global Arrays uses a *memory allocator* library [7] to store the global data objects and the data used in communication. The user only has to start the *memory allocator* with sufficient sizes for `MA_STACK` and `MA_HEAP`, but the description in which way *Global Arrays* uses the *memory allocator* is not very detailed, so usually the user has to guess what sizes to choose. There is an example program for *Global Arrays* where the user can see, how the *memory allocator* is started and we just tried changing the example. We could not find that the performance was influenced significantly by the sizes and so we usually choose the sizes as large as possible to solve problems as large as possible.
Apart from the mystery of the *memory allocator* the usage of *Global Arrays* is relatively simple. The data are partitioned to large contiguous blocks and each block is assigned to one process. With the routine `GA_DISTRIBUTION` the user can find out which rows and which columns are assigned to the current process. But to use *Global Arrays* it is not even necessary to know which process owns which part of the data, the user can create the matrix for example column-wise and put one column after the other to the global matrix with the help of the routine `GA_PUT`. Since this causes a lot of communication it will be better to create a larger block of the matrix in each process and put it to the global matrix. If each process creates and puts to the global matrix only its local part of the matrix no communication at all has to be done.
Another way, if there is enough memory, is to create a local copy of the whole matrix on one node (or one column block after the other on one node) and call `GA_PUT` on this node only.
The calling sequences of the computational routines are much simpler than the ones of the sequential *LAPACK* routines as matrix dimensions and all details of the distribution are included in the global data object. This also means that it is not possible to call a `GA` routine with a submatrix of a global matrix. In that case the submatrix has to be copied to a new global matrix.
The symmetric eigensolver of *Global Arrays* is much less flexible than the one from *ScaLAPACK* as there is no choice to compute eigenvalues only or only a few eigenvectors. In case of failure there is no information on what failed and the user cannot require less accurate eigenvalues and perhaps non-orthogonalized eigenvectors.
The calling sequence for `GA_DIAG_STD` is the following:

```
! Call GA_DIAG_STD with the global matrix g_a
CALL GA_DIAG_STD(g_a,g_c,EVALS)
```

where `g_a` contains the global symmetric input matrix, `g_c` returns the global eigenvector matrix and `EVALS` is a local array which on each node contains all computed eigenvalues.
To use *Global arrays* and to set up the global matrix `g_a` the user has to do at least the following steps:

```
! Initialize the Memory Allocator package
STATUS=MA_INIT(MT_DBL,MA_STACK,MA_HEAP)
! Create global array g_a
STATUS=GA_CREATE(MT_DBL,N,N,'A',CHUNK1,CHUNK2,g_a)
! Get my processor identification
ME = GA_NODEID
! Allocate a local matrix and fill it with data
! It is useful to fill only those parts of the global
! matrix from the local processor which will be stored there
! Find out which part of the global matrix this node has
! (optional)
CALL GA_DISTRIBUTION(g_a,ME,ILO,IHI,JLO,JHI)
! Process ME has rows ILO to IHI and columns JLO to JHI
! Allocate local array A to fill with matrix data
ALLOCATE(A(IHI-ILO+1,JHI-JLO+1))
! Fill local matrix with global data
DO J=1, JHI-JLO+1
   DO I=1, IHI-ILO+1
     A(I,J)= Function of global indices ILO+I-1,JLO+J-1
   ENDDO
ENDDO
! Put local matrix to global matrix,
! global starting position is ILO, JLO,
! global end position is IHI, JHI,
! leading dimension of local matrix A is IHI-ILO+1
CALL GA_PUT(g_a, ILO, IHI, JLO, JHI, A, IHI-ILO+1)
! Deallocate local matrix
DEALLOCATE(A)
```

2.3.4 Comparison of the Different Libraries

The main advantage of *ScaLAPACK* is the great flexibility of PSSYEVX. Thus if only a part of the eigensystem is needed there is no choice which library to use. You can only call PSSYEVX from *libsci*/*ScaLAPACK* with data created directly to match the distribution of *ScaLAPACK* or use the input/output and distribution routines from *NAG Parallel Library* to call PSSYEVX.
Global Arrays offers the interface to the *PeIGS* library but there is no flexibility at all. It can only handle the full symmetric eigenvalue problem and compute all eigenvalues and all eigenvectors.

3 Basics of Performance Measurements

For this and the following sections the letter n will be used to indicate matrix sizes, i.e. $n = 400$ means that the eigenvalues and eigenvectors of a 400×400 matrix are computed. The letter np means the number of processors and nb the block size for the routines PSSYEVX and PSSYEV.

3.1 Used Systems

All measurements of MPP routines (unless mentioned something else) were done on a CRAY T3E-900 (i.e. 450 MHZ or 900 MFLOPS peak performance) with 256 nodes with 128 MByte memory per node, a data and instruction cache of eight KByte each, a level-2 cache for data and instructions of 96 KByte, and stream

buffers turned on. Version 3.2 of the computing environment was used. Additional measurements were done on a CRAY T3E-1200 with 256 nodes with 512 MByte memory per node and the same computing environment.
The measurements of the PVP routines were done on a CRAY T90 with ten processors (each with 1.8 GFLOPS peak performance) and 1024 MWords corresponding to eight GByte of main memory, a skalar data cache of 1024 W = 8 KByte, eight vector registers 128 words each and 512 memory banks. To allow multiprogrammin in John von Neumann-Institute for Computing (NIC) the user only has access to 256 MWords = 2 GByte of main memory per job. The used computing environment was the same as for the CRAY T3E-900.

3.2 Problems Studied

We tried to test examples of many different sizes and to study a couple of processor grids on CRAY T3E to find out as many influencing factors as possible.
On CRAY T3E we examined square processor grids with $np = 4, 16, 25, 36,$ and 64 nodes as well as rectangular grids with 8 and 32 processors. The matrix sizes for the *ScaLAPACK* and *Global Arrays* varied from problems with about 200×200 matrix elements per node (see Table 1) to the maximum size possible with the corresponding number of processors for up to 32 processors. For 36 and 64 processors and for `F02FQFP` on all numbers of nodes we only measured the small and the large problem as mentioned in Table 1 and a few intermediate matrix sizes. For smaller numbers of processors we measured many matrix sizes n, especially complete sequences of all n divisible by 100 and n divisible by 64 between $n = 1200$ and $n = 2000$.
For the *ScaLAPACK*-routines we studied the following block sizes:
$nb = 5, 10, 16, 20, 25, 30, 32, 40$ for $n = 400$ on 4 nodes and
$nb = 10, 16, 20, 25, 32, 50$ for $n = 500$ on 4 nodes.
Execution times were found to be decreasing for $5 \leq nb \leq 16$ and increasing for $nb \geq 20$ for those small problem sizes. So we measured *ScaLAPACK*-routines with block sizes of $nb = 16$ and $nb = 20$ for $400 \leq n \leq 2600$ and $nb = 20$ and $nb = 32$ for $n \geq 3000$. For a few large problems $nb = 32$ delivered the fastest execution time.

The problem sizes for the PVP routines varied from to 400 to 2000, except for `SSYEVD` on CRAY T3E: due to lack of memory the maximum n was 1856.

To test the correctness of the solutions we constructed the matrices in the following way: For an $n \times n$ test matrix we randomly choose n eigenvalues in the interval [-10, 2000] with the Fortran 90 intrinsic function `RANDOM_NUMBER` and appropriate scaling.
In order to also analyze what happens if there are clusters of eigenvalues we also choose a case where 7 of those eigenvalues were taken multiple times, namely 4, 5, 11, 15, 20, 150, and n-267 times the same eigenvalue. The multiple eigenvalues then were modified randomly by values in the range of [-1.e-5,1.e-5] be-

cause exactly equal eigenvalues gave strange results with PSSYEV (see details of the routines examined, section 2.2). Then we choose another random vector of length n, scaled it to have norm 1 and computed a Householder transformation of the diagonal matrix containing the eigenvalues as diagonal elements. So we got a full symmetric matrix of which we knew all the eigenvalues and -vectors. The maximum deviation of the computed eigenvalues and the "input" eigenvalues was also measured and an orthogonality test for the eigenvectors was done, too.
To show the power of F02FQFP we should have included diagonally dominant test matrices.

3.3 Performance Analysis Tools Used

Real time was measured using the intrinsic function system_clock, the elapsed CPU-time using the Cray function second. On CRAY T90 the returned value for the CPU-time is the time accumulated by all processes in a multitasking program, including wait-semaphore time.

On CRAY T3E, we used the performance analysis tool PAT [12] from Cray Research Inc.. PAT provides a fast, low overhead method for estimating the amount of time consumed in procedures, determining load balance across processing elements (PEs), generating and viewing trace files, timing individual calls to routines, and displaying hardware performance counter information. We used PAT to get information on the BLAS and communication routines that contribute to the execution time of the different codes and to get the number of floating point operations to determine the MFLOPS. The MFLOPS given by PAT were sometimes wrong. PAT only gives information for the whole program, but especially for higher matrix dimensions, only the subroutine call contributes to the results. The values for the MFLOPS were checked via PCL [15], which allows measurements for the subroutine call only.

On CRAY T90, the Hardware Performance Monitor (hpm) and Perftrace [14] were used for performance measurements. The hpm command gives information for the whole program only; Perftrace gives more detailed informations, but does introduce overhead. Additionally, Perftrace only works with multitasked codes if the environment variable $NCPUS is set to 1. We used hpm in order to get information about the MFLOPS and the average number of concurrent CPUs. The average number of concurrent CPUs indicates the level of parallelization of an algorithm measured in a dedicated environment. The results from hpm were checked against those of Perftrace.

4 Factors that Influence MPP Performance

There are many factors that influence the performance on MPP systems, some of them also influence performance on other machines, some are specific to the message-passing programming paradigm.

In the tables of the following sections and in the appendix we will distinguish between eigensystems with no large clusters of eigenvalues and with one large cluster of eigenvalues and between "small" and "large" problems, where small and large is related to the number of matrix elements per processor. So a small problem means that each processor has about 200×200 elements of the matrix whose eigenvalues and eigenvectors are to be computed and a large problem means that each processor has at least about 1000×1000 matrix elements. Table 1 shows the actual matrix dimensions of small and the minimal dimensions of large problems on different numbers of processors.

Table 1: Sizes of small and large problems on different numbers of processors

	4 nodes	8 nodes	16 nodes	25 nodes	32 nodes	36 nodes	64 nodes
small problem	$n =$ 400	$n =$ 512	$n =$ 800	$n =$ 1000	$n =$ 1152	$n =$ 1200	$n =$ 1600
large problem	$n =$ 2000	$n =$ 2816	$n =$ 4000	$n =$ 5000	$n =$ 5632	$n =$ 6000	$n =$ 8000

With problems larger than about $n = 4800$ a problem arose with PSSYEVX. As mentioned in the *ScaLAPACK* Users' Guide [1] *ScaLAPACK* uses "a nonscalable definition of clusters" (to remain consistent with *LAPACK*). "Hence, matrices larger than $n = 1000$ tend to have at least one very large cluster." So even for equally spread eigenvalues the arising clusters were too large to be reorthogonalized on one processor when we took the default value of 1.E-3 for ORFAC. We had to choose ORFAC=1.E-4 instead. This lead to a slightly higher value of the residual $||Q^T Q - I||/(ulp * n)$ where Q means the matrix of the computed eigenvectors, I the identity matrix, $||.||$ the 1-norm ($\max_{j=1,n} \sum_{i=1,n} |a(i,j)|$), n the matrix size, and $ulp = 4.44089209850062616E-16$ is the relative machine precision times base of the machine. This residual was approximately 5.1452E-2 for $n = 4900$, $np = 32$, and $nb = 32$ with ORFAC=1.E-3 and 0.16214 with ORFAC=1.E-4 for example. We could also see that the BLAS 1 routines SDOT and SAXPY occurred in the statistics of routines used when we used the default ORFAC. This is due to the reorthogonalization taking place although we didn't want to create clusters. Therefore, performance data and percentage of BLAS use were measured with ORFAC=1.E-4 for $n \geq 4000$ in the case where a matrix with equally spread eigenvalues was constructed.

In the case of one large cluster ORFAC had to be as small as 1.E-11 in order to be able to compute problems of $n > 2000$ for $np = 4$ and then $||Q^TQ - I||/(ulp * n) = O(10^6)$ which means $||Q^TQ - I||/n = O(10^{-9})$. Thus the eigenvectors still are nearly orthogonal to an accuracy which might be sufficient in many cases. For ORFAC small enough or ORFAC=0 the performance of the problem with one large cluster of eigenvalues is the same as for equally spread eigenvalues.
The largest problems with one large cluster that could be computed on our machine (128MB memory per node) with the default ORFAC were: $n = 2000$ for $np = 4$, $n = 2240$ for $np = 8$, $n = 2500$ for $np = 16$, $n = 2624$ for $np = 25, 32,$ and 36, and $n = 2700$ fo $np = 64$.

4.1 Usage of BLAS Routines

All library routines examined use BLAS routines for single node computations, hence vendor optimized BLAS routines, here those from *libsci*, are a very important factor to improve performance. Due to the small level 1 cache on CRAY T3E and BLAS 1 (vector-vector) routines slowing down significantly when data are not in level 1 cache, for all but very small problems performance of BLAS 1 routines is very poor. BLAS 2 (matrix-vector) routines still cannot deliver high performance, so it is preferable to use BLAS 3 (matrix-matrix) routines because cache may be reused effectively here.
ScaLAPACK routines use blocked algorithms to allow the usage of BLAS 3 routines, whereas *PeIGS* and the Jacobi solver from *NAG Parallel Library* rely heavily on BLAS 1 routines. This can be seen from Tables 2 and 3.
The percent values for small problems were taken from all numbers of processors, and the smaller values arise from the larger numbers of processors as communication costs increase with the number of processors (see section 4.2). For GA_DIAG_STD the small values for SDOT belong to the large values for SAXPY and vice versa, where the percent value for SDOT is decreasing with the number of nodes.
The percent values for the large problems are taken from all numbers of processors and all problem sizes larger than or equal to those mentioned in Table 1.
The values for PSSYEVX and large problems are taken from the case of 4 processors and $n = 2000$ as all the other large problems were too large for reorthogonalization.

It can be seen that PSSYEVX is best optimized for BLAS 3 usage if there are no large clusters of eigenvalues. The high percent value of BLAS 1 SROT usage in PSSYEV is due to the QR-algorithm to compute the eigenvalues of the tridiagonal matrix. As this is done on all processors simultaneously it contributes completely to the average BLAS 1 use. If only one processor would do the QR-algorithm only $\frac{1}{np}$ would be BLAS 1 and $\frac{np-1}{np}$ would be communication/wait time.

For F02FQFP, the values for SROT and SDOT decrease with the number of processors involved, the values for SSWAP increase. The values in Table 3 for the large

Table 2: Percentage of time spent in different BLAS routines: no large clusters of eigenvalues

percentage of time spent in		small problem	large problem
PSSYEVX	BLAS 3 SGEMM	6 - 8%	$\geq$ 30 %
	BLAS 2 SGEMV	7 - 8 %	$\approx$ 27 %
	BLAS 1	-	$\leq$ 1 %
PSSYEV	BLAS 3 SGEMM	3 - 4 %	$\geq$ 9 %
	BLAS 2 SGEMV	2 - 5 %	7 - 9 %
	BLAS 1 SROT	26-33 %	$\geq$ 56 %
GA_DIAG_STD	BLAS 2, 3	-	-
	BLAS 1 SDOT	28-41 %	39-58 %
	BLAS 1 SAXPY	7 -8 %	19-27 %
F02FQFP	BLAS 2, 3	-	-
	BLAS 1 SROT	37-45 %	41-49 %
	BLAS 1 SDOT	19-27 %	19-29 %
	BLAS 1 SSWAP	9-14 %	12-17 %

Table 3: Percentage of time spent in different BLAS routines: one large cluster of eigenvalues

percentage of time spent in		small problem	large problem
PSSYEVX	BLAS 3 SGEMM	$\approx$ 6 %	$\approx$ 5 %
(4 nodes only)	BLAS 2 SGEMV	$\approx$ 5 %	$\approx$ 5 %
	BLAS 1	$\approx$ 7 %	$\approx$ 21 %
PSSYEV	BLAS 3 SGEMM	3 - 6 %	$\geq$ 10 %
	BLAS 2 SGEMV	3 - 4 %	9 -11 %
	BLAS 1 SROT	21-24 %	$\geq$ 52 %
GA_DIAG_STD	BLAS 2, 3	-	-
	BLAS 1 SDOT	25-38 %	40-55 %
	BLAS 1 SAXPY	6 -10 %	19-33 %
F02FQFP	BLAS 2, 3	-	-
	BLAS 1 SROT	25-32 %	20-28 %
	BLAS 1 SDOT	24-27 %	6-21 %
	BLAS 1 SSWAP	17-20 %	41-56 %

problem are only valid, if the number of columns of the matrix on each processor is even. If this number is odd, the values are about 30 % for `SROT`, 20 % for `SDOT`, and 30 % for `SSWAP`.

4.2 Load Balance and Communication Overhead

Load balance is a very important factor for MPP performance. If the load is not properly balanced a lot of time is spent waiting for other processors to finish computation and send data needed to continue.
An extreme example is `PSSYEVX` in the case of one large cluster. There is

only one processor with the highest number of operations in Table 9[1], all the other processors have operation counts in the same order of magnitude as the smaller value. This means that only one process is doing computations (the reorthogonalization) most of the time and all the others spend more than 50 % of the time in communication which in this case means waiting (see Table 7). As PAT only delivers percentage of communication of the whole application, i.e. an average over all nodes, the time spent waiting on the $np - 1$ nodes is even higher than indicated in Table 7.

Tables 6 and 7 show for all routines that the communication costs increase as the number of processors gets higher and decrease as the problem size increases.
PSSYEV has the lowest percentage of communication costs, especially for large problems with one large cluster of eigenvalues. This seems to be due to a very good load balance as can be seen when looking to the operation counts in Tables 8 and 9.
The relative communication costs in GA_DIAG_STD are always higher than in the other routines and the operation counts are in a wider range except for the case with one large cluster and PSSYEVX.
The imbalance of operations decreases with the matrix size for the *ScaLAPACK* routines (e.g. on 16 nodes the node with the largest operation count for PSSYEVX has about 18 % more operations than the one with the smallest operation count for the small problem and only about 8 % more for the large problem. The corresponding values for PSSYEV are 20 % for the small problem and 4 % for the large one). For GA_DIAG_STD the load imbalance increases with the problem size, e.g. on 16 nodes and the small problem the node with the largest operation count has about 65 % more operations than the one with the smallest count and for the large problem it has 70 % more operations to do.
If there is one large cluster of eigenvalues the load imbalance of GA_DIAG_STD increases probably due to the orthogonalization of eigenvectors which perhaps is not too well parallelizable. The largest operation count here is more than twice as high as the smallest one in most cases. Communication costs (waiting for other nodes to finish) therefore are seldom less than 20 % of the whole computation costs with GA_DIAG_STD.
For F02FQFP, the load balancing is good, if the number of processors is a divisor of the matrix size. That is the case for most of the test calculations. Only the small problem on 36 processors shows a significant imbalance (see Tables 8 and 9): for the matrix size $n = 1200$, $[1200/36] = 34$ columns of the matrix are stored on each processor except the last, which only gets 10 columns.

[1]Tables 6 to 9 can be found in the appendix

4.3 Matrix Distribution

ScaLAPACK distributes the matrices in some block-cyclic 2-dimensional way where the block sizes are an important performance factor. For detailed information about the data distribution see the *ScaLAPACK* Users' Guide. Experiments with different block sizes have to be made to find out an optimal or nearly optimal block size.
ScaLAPACK also allows the user to choose the shape of the processor grid, e.g. 2×4 or 4×2 processors, which also sometimes influences performance. Usually a square grid gives best performance, but for non square grids there was not always found the optimal grid shape for a given number of nodes. For example with 8 nodes and block sizes of $nb = 16$ and $nb = 20$ almost always a 2×4 processor grid gave the shortest execution time whereas for $nb = 32$ the shortest execution time was reached with a 4×2 processor grid.
Routine `F02FQFP` from *NAG Parallel Library* assumes that the matrix is distributed block columnwise.
Global Arrays only offers the chance to choose the minimum block size in each direction, so you can choose column block distribution if you force the row block size to be at least the matrix dimension or row block distribution by forcing the column dimension to be at least the matrix dimension. The default distribution is as square as possible. *Global Arrays* then uses the simple distribution to contiguous blocks and redistributes the matrices to the storage scheme which is used in *PeIGS*. This is a one-dimensional distribution where complete columns are assigned to each node. Since the redistribution is done in the routine `GA_DIAG_STD`, the user has no influence on the assignment of colums to processors.

4.4 Block Sizes

Only for the routines from *ScaLAPACK* or *libsci* respectively the user can choose block sizes, therefore everything said now only concerns `PSSYEV` and `PSSYEVX`.
As mentioned above (see section 3.2) best performance was found for block sizes between $nb = 16$ and $nb = 20$ for most problems and for $nb = 32$ for some of the larger problems. In most cases the difference between $nb = 20$ and $nb = 16$ or $nb = 32$ was very small and we could not find a strict rule which block size should be preferred.
The block sizes often slightly influence load balance, especially in cases where $\frac{n}{nb\sqrt{np}}$ is integer for one block size and is not for the other one. For example with $n = 400, nb = 20, np = 4$ each processor has a 200×200 matrix in its memory whereas with $n = 400, nb = 16, np = 4$ processor $(0,0)$ has a 208×208 matrix, processor $(0,1)$ has a 192×208 matrix, processor $(1,0)$ a 208×192 and processor $(1,1)$ a 192×192 matrix. As a result, the operations for `PSSYEV` without large clusters vary between $175 \cdot 10^6$ and $184 \cdot 10^6$ for $nb = 20$ (see Table 8) and $170 \cdot 10^6$ and $195 \cdot 10^6$ for $nb = 16$. As expected, load balance is better for $nb = 20$, com-

munication costs are lower (19 % versus 23 %), and performance is slightly better (2.3057 sec versus 2.3669 sec).
PSSYEV on 4 nodes seems to be sensitive for block sizes for problems up to $n = 1900$ as $nb = 20$ only delivered the better performance for $n = 400, 800, 1100, 1200,$ and 1500, in all other cases $nb = 16$ was slightly faster. For larger numbers of processors it was no longer as significant as with 4 processors, but the differences in general were negligible.
With both routines we got a large degradation in performance on 4 nodes for $n = 2048$ and $nb = 16$, (see Figure 1). We found this phenomenon again on 16 nodes with $n = 4000$ to 4160 and $nb = 32$ on 25 nodes for $n = 4900$ to $n = 5200$, on 36 nodes for $n = 6000$ and on 64 nodes for $n = 8000$, always with block size $nb = 32$.
With the help of the performance tool Apprentice[13] we could find out that a

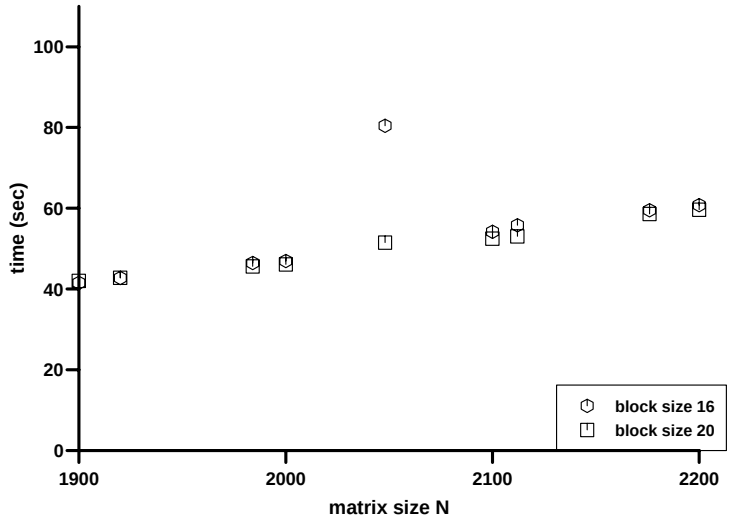

Figure 1: Execution times for PSSYEVX on 4 nodes, different block sizes, computation of all eigenvalues and eigenvectors: no large clusters of eigenvalues

lot of time was spent in SGEMM in the back transformation of the computed eigenvectors. SGEMM was called with the second matrix transposed and the first one not, and the matrix size was 1024×1024. This is just the size of at least one of the local matrices in the cases mentioned above.
Performance measurements for SGEMM showed that with this size and only in the case with second matrix transposed - first one not - there is a degradation of performance of a factor of about 9. This is due to a performance problem of SGEMM from *libsci*. Called with random matrices the time for SGEMM in the above situation is 4.6 sec for $n = 1000$, 54.8 sec for $n = 1024$ and 6.1 sec for $n = 1050$, hence it takes almost 9 times as long to multiply two 1024×1024 matrices - only the second one transposed - than to multiply two 1050×1050 matrices.

5 Performance of Parallel Codes on CRAY T3E

Two diagrams with performance results will be shown. Different libraries are compared and different numbers of processors are used. For each routine and each number of processors the configuration (e.g. grid shape, block size) with the shortest execution time is shown in the diagrams. Figures 2 and 3 show the execution times for the computation of all eigenvalues and all eigenvectors of a real symmetric matrix of size $n = 2000, \ldots, 2500$.

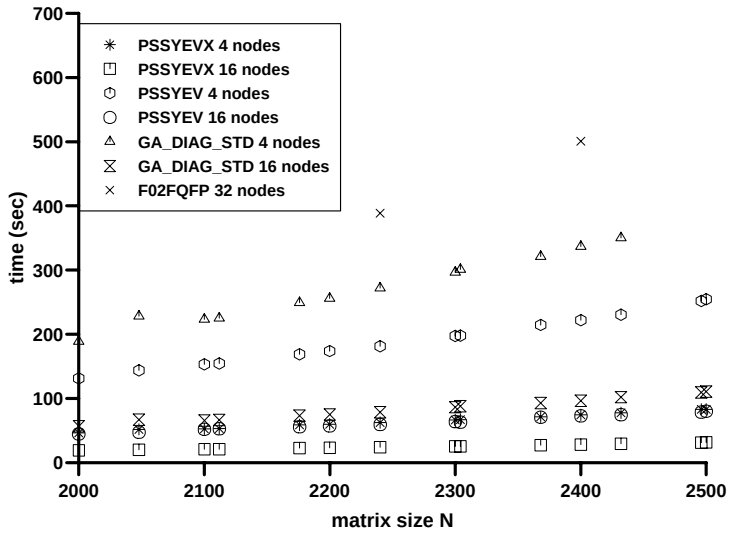

Figure 2: Execution times of the different routines on different numbers of nodes, computation of all eigenvalues and eigenvectors: no large clusters of eigenvalues

Figure 2 shows the times in the case where the eigenvalues are equally spread and reorthogonalization is not necessary. It can be seen that for equally spread eigenvalues PSSYEVX on 4 nodes is as fast as PSSYEV on 16 nodes. One reason, of course is, that PSSYEV needs twice as many operations as PSSYEVX. The other reason is, that the sequential QR-algorithm within PSSYEV uses a lot of BLAS 1 routines and consequently reaches less MFLOPS per node than PSSYEVX.
Although GA_DIAG_STD uses the same algorithm as PSSYEVX if there are no large clusters of eigenvalues the number of operations is much higher even for the node with the smallest number of operations (see Table 8). Due to the poor load balance of GA_DIAG_STD, see Table 8, it is even higher than that of PSSYEV on the node with the highest operation count. For large problems also the MFLOPS per node reached with GA_DIAG_STD are significantly lower than the ones reached by PSSYEV or PSSYEVX, mainly because it is completely based on BLAS 1 routines and therefore performance is additionally reduced by cache misses.
The Jacobi algorithm F02FQFP of *NAG Parallel Library* needs about 4 to 5 times as

many operations as GA_DIAG_STD or PSSYEV for the general test matrices we used so it cannot be competitive. Nevertheless we show execution times for F02FQFP on 32 nodes. It can be seen that it is slower on 32 nodes than the slowest other routine on 4 nodes.

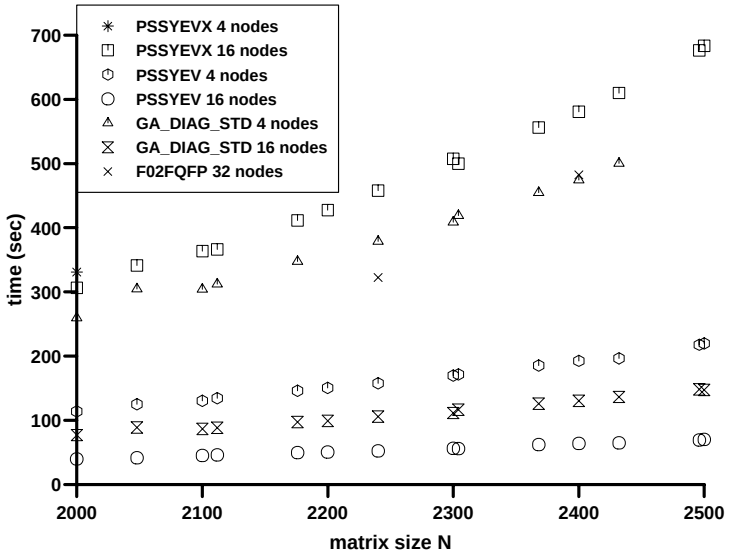

Figure 3: Execution times of the different routines on different numbers of nodes, computation of all eigenvalues and eigenvectors: one large cluster of eigenvalues

Figure 3 shows the execution times for a matrix with one large cluster of $n - 267$ eigenvalues. In PSSYEVX and GA_DIAG_STD the eigenvectors belonging to this cluster are reorthogonalized.

Now on four nodes the largest problem that could be solved with PSSYEVX was $n = 2000$. The execution times for PSSYEVX remain almost constant with a larger number of nodes. Only the problem sizes that can be solved become somewhat higher (see section 4).

It can be seen that the execution times for PSSYEV and F02FQFP are slightly lower in the case of one large cluster of eigenvalues than in the case of equally spread eigenvalues. The execution times of GA_DIAG_STD on the other hand become higher as eigenvectors are reorthogonalized. Consequently the difference between PSSYEV and GA_DIAG_STD becomes larger than in the case with no large clusters. F02FQFP on 32 nodes now for small problems is a little faster than GA_DIAG_STD on 4 nodes and PSSYEVX on 16 nodes.

For the solution of the symmetric eigenvalue problem there is always one *ScaLAPACK* routine with highest performance: if eigenvalues are not clustered this is PSSYEVX, for one large cluster of eigenvalues whose eigenvectors have to be reorthogonalized PSSYEV is the fastest routine.

5.1 Scalability

In Figures 4, 5, and 6 speedup diagrams for *ScaLAPACK* and *Global Arrays* are shown, execution times for different numbers of nodes are compared to 4 nodes. The comparison of 4-nodes run times to the run times of the best sequential codes will be presented in section 6.3.

The problem sizes are relatively large for 4 nodes but relatively small for more than 20 nodes. It can be seen that for 8 nodes the speedup values remain almost constant through the whole range of n, hence the problems are large enough for 8 nodes. For all other numbers of nodes increasing speedup values can be seen.
In Table 4 the maximum speedup values for one number of nodes compared to another one can be seen. For example the speedups of 16 nodes compared to 8 nodes are usually higher than the quotient of the maximum speedup of 16 nodes to 4 nodes divided by the speedup of 8 nodes to 4 nodes as they are reached at a problem size that is too large to be computed on 4 nodes with 128 MB memory.
Rather good speedup values are reached with all routines if problem sizes are large enough.

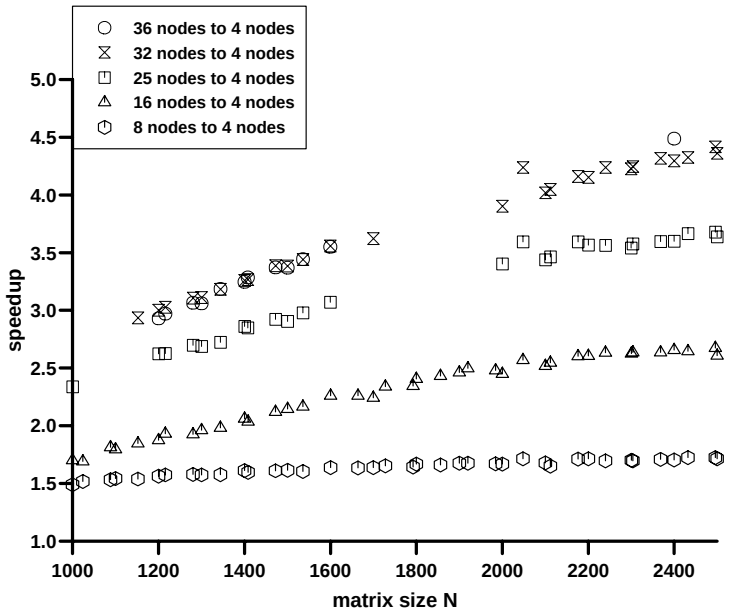

Figure 4: Speedups of PSSYEVX, different numbers of nodes versus 4 nodes, computation of all eigenvalues and eigenvectors: no large clusters of eigenvalues

PSSYEV scales slightly better than PSSYEVX. As the only difference between the two routines is in the second step, the computation of eigenvalues and eigenvectors of the tridiagonal matrix, this can only be due to a good scalability of the computation of eigenvectors in PSSYEV.

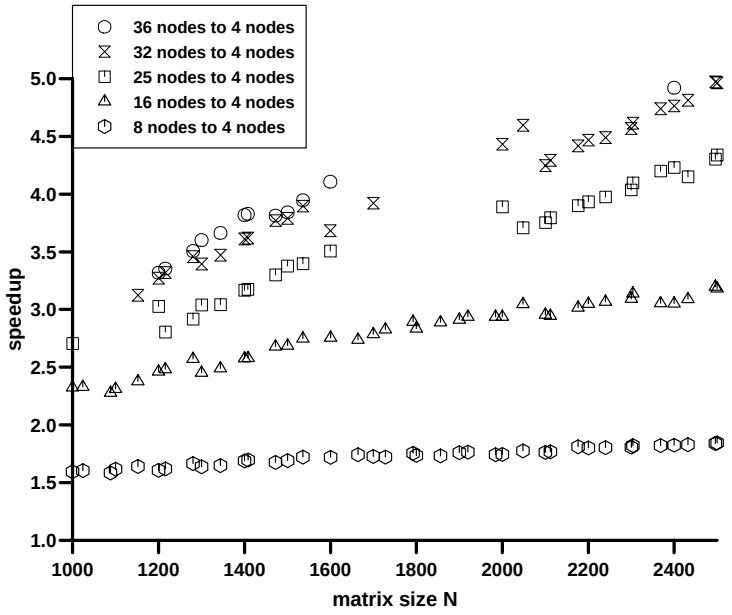

Figure 5: Speedups of PSSYEV, different numbers of nodes versus 4 nodes, computation of all eigenvalues and eigenvectors: no large clusters of eigenvalues

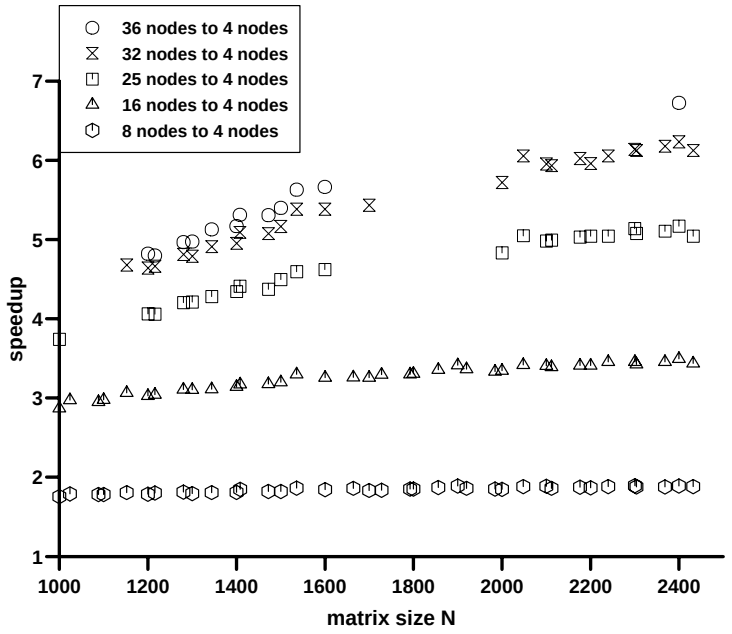

Figure 6: Speedups of GA_DIAG_STD, different numbers of nodes versus 4 nodes, computation of all eigenvalues and eigenvectors: no large clusters of eigenvalues

Table 4: Maximum speedups reached with different libraries

number of nodes to number of nodes	8:4	16:4	16:8	25:8	32:8	32:16	36:16
PSSYEVX, no large clusters	1.76	2.68	1.63	2.36	2.71	1.79	1.70
PSSYEV, no large clusters	1.85	3.20	1.81	2.53	2.95	1.73	1.90
PSSYEV, one large cluster	1.83	3.15	1.82	2.53	2.92	1.75	1.92
GA_DIAG_STD, no large clusters	1.89	3.49	1.88	2.86	3.51	1.93	2.10
GA_DIAG_STD, one large cluster	1.99	3.67	1.88	2.72	3.45	1.89	2.08

`GA_DIAG_STD` scales better than both other routines although load balance is worse. As `GA_DIAG_STD` is slower than the other routines in most cases, the better scalability does not make it superior.

The diagrams only show speedups in the case of no large clusters of eigenvalues. In the case of one large cluster of eigenvalues `PSSYEVX` almost does not scale at all, the speedup values are in the range of 1.05 to 1.15 for all numbers of nodes compared to 4 nodes.
The scaling of `PSSYEV` and `GA_DIAG_STD` is approximately the same in both cases.

Table 5: Speedup values for F02FQFP, matrix size n=1600

	no large clusters of eigenvalues		one large cluster of eigenvalues	
number of processors	execution time in sec	speedup	execution time in sec	speedup
1	3379		3143	
4	951	3.6	845	3.7
8	507	6.7	439	7.2
16	267	12.7	228	13.8
32	139	24.3	122	25.8
64	75	45.1	80	39.3

The execution times for `F02FQFP` are much higher than for the other codes, but the scalability of the algorithm, even for a relative small matrix size, is very good as can be seen from Table 5.
The speedup values for the problem with one large cluster of eigenvalues are higher than those for the problem with no large clusters of eigenvalues, except for the run on 64 processors: the number of columns on each processor is odd whereas for all other numbers of processors this value is even.

5.2 New Algorithm and New Machine

After almost all the measurements were done the CRAY T3E-900 was changed to a CRAY T3E-1200 with 600 MHZ clock rate and 1200 MFLOPS peak performance

and 512 MByte memory per node. Control measurements showed that on average PSSYEVX ran 1.13 times faster on the new machine than on the old one.
In the meantime there exists a new algorithm for the reduction phase in *ScaLAPACK*, using less communication by combining messages. There is a new public domain version of PSSYEVX available using the new reduction algorithm which may be received via www from **`http://www.cs.utk.edu/~kstanley/`**. It can be seen that especially for relatively small problems on many nodes (i.e. small matrix parts per node) the performance gain is high if no large clusters of eigenvalues occur. The new reduction can make the whole computation almost twice as fast on 36 nodes for $n = 400$. The speedup decreases with the problem size and is only about 1.1 for $n = 6000$ on 36 nodes or almost 1.0 for $n = 4800$ on 16 nodes. The speedup on 16 nodes of the new algorithm compared to the old one is shown in Figure 7.

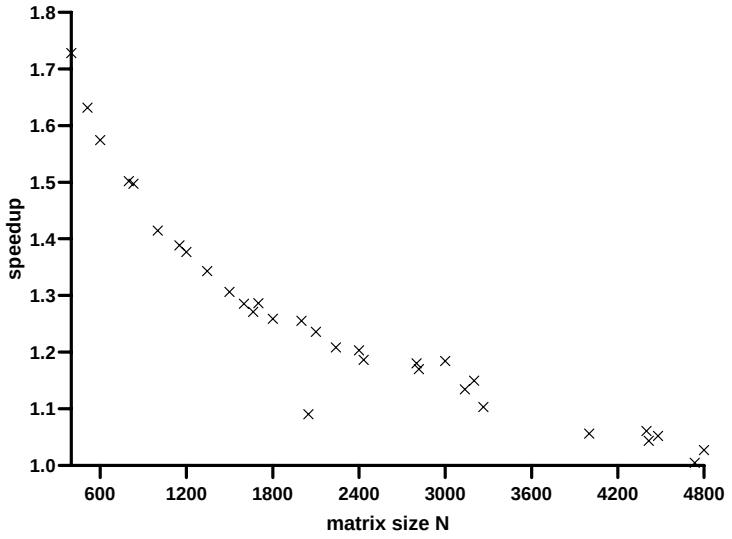

Figure 7: Speedup of PSSYEVX with new reduction algorithm versus old algorithm, 16 nodes on T3E-1200, computation of all eigenvalues and eigenvectors: no large clusters of eigenvalues

If there is one large cluster of eigenvalues and reorthogonalization is necessary, the speedup is very small because the part of the reduction phase in the whole computation is so small in this case.
The new reduction algorithm may be used with PSSYEV as well, but the interface is not ready up to now. The absolute performance gain will be the same but the relative gain, i.e. speedup will be smaller as the computation of eigenvalues and eigenvectors of the tridiagonal matrix takes a larger part of the computation in this case.

6 Performance of Single Node Codes on CRAY T3E

The following diagrams show execution times achieved for the single node codes described in section 2.2.2.
The percentages of BLAS use on CRAY T3E were found with the help of the performance analysis tool *PAT* [12] which is explained in more detail in section 3.3. For $n = 1024$ the performance of all single node routines on CRAY T3E was significantly worse than for other problem sizes and the percentage of `SGEMM` usage was higher due to the performance breakdown of *libsci*'s `SGEMM` for this size (see section 4.4).

6.1 No Large Clusters of Eigenvalues

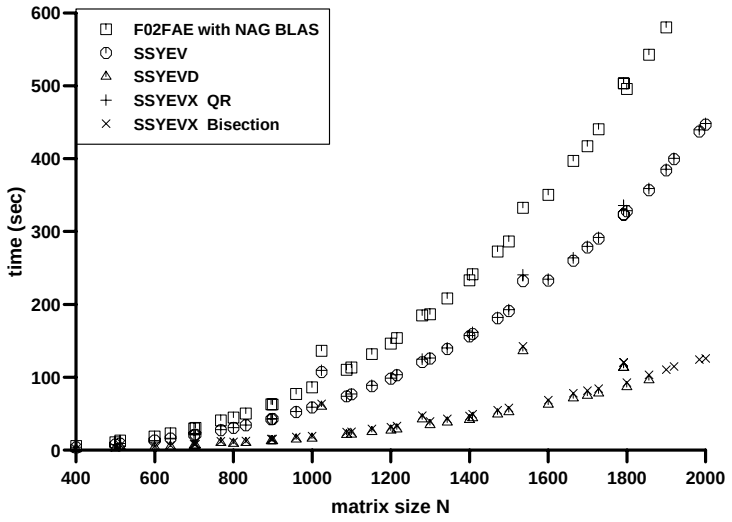

Figure 8: Execution times for different codes, computation of all eigenvalues and eigenvectors on CRAY T3E: no large clusters of eigenvalues

Figure 8 shows the execution times for problems with no large clusters of eigenvalues. The highest execution time is needed by `F02FAE` from *NAG* with *NAG* BLAS, `F02FAE` with BLAS from *libsci* gives the same results as `SSYEV`. The best results show `SSYEVX` (bisection) and `SSYEVD`. 128 MByte memory per node were not sufficient to give results for `SSYEVD` for matrix sizes greater than 1856.

`SSYEVD` and `SSYEVX` (bisection) show the most exact eigenvalues but need the shortest execution time. This is due to the fact, that `SSYEV` spends about 80 % of its time in `SROT` and only 8 % in `SGEMM`, 9 % in `SSYMV` and `SGEMV`, whereas `SSYEVD` spends about 50 % in `SGEMM`, 35 % in `SSYMV` and `SGEMV` and `SSYEVX` (bisection) about 37 % in `SGEMM`, 34 % in `SSYMV` and `SGEMV`.
For the measured matrix sizes `SSYEVD` reaches very good values for MFLOPS, up

to 37 % of the peak performance (i.e. about 330 MFLOPS), SSYEVX (bisection) up to 28 % , SSYEV only 17 %. Good performance values can only be achieved by the extensive use of optimized BLAS 3.

6.2 One Large Cluster of Eigenvalues

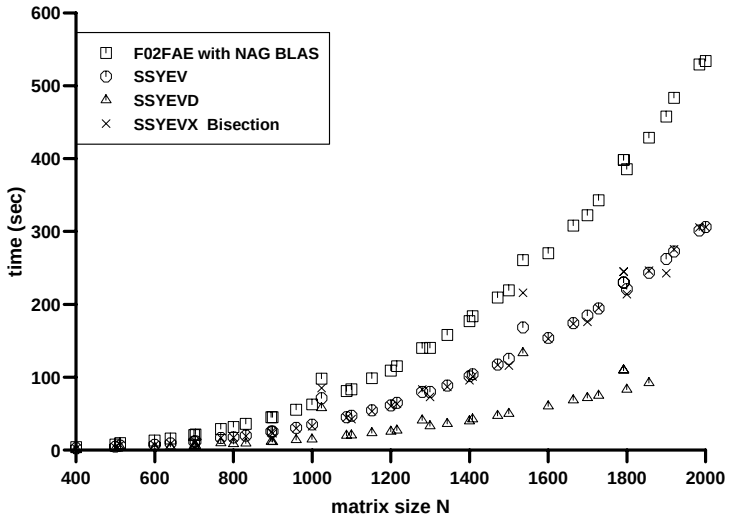

Figure 9: Execution times for different codes, computation of all eigenvalues and eigenvectors on CRAY T3E: one large cluster of eigenvalues

Figure 9 shows the execution times for problems with one large cluster of eigenvalues. As for the nonclustered eigenvalues, the highest execution time is needed by F02FAE from *NAG* with *NAG* BLAS. F02FAE with BLAS from *libsci* gives the same results as SSYEV and SSYEVX (QR), since the algorithms are identical. In contrast to the test case with no large clusters of eigenvalues, SSYEVX (bisection) has execution times similar to those of SSYEV. SSYEVD needs the lowest execution time and additionally shows the most exact eigenvalues.
SSYEV spends about 70 % of its time in SROT and only 12 % in SGEMM, 13 % in SSYMV and SGEMV, whereas SSYEVD spends about 50 % in SGEMM, 35 % in SSYMV and SGEMV and SSYEVX (bisection) about 15 % in SGEMM, 75 % in SSYMV and SGEMV.
For the measured matrix sizes SSYEVD reaches very good values for MFLOPS, up to 35 % of the peak performance (about 315 MFLOPS), but SSYEVX (bisection) only up to 22 % and SSYEV up to 19 %.

6.3 Comparison with Parallelized Codes

Figures 10 and 11 show the best two sequential routines on CRAY T3E compared to the two *ScaLAPACK* routines on 4 nodes. If there is no large cluster of eigen-

values PSSYEVX on 4 nodes is faster than the fastest sequential routine SSYEVD.

In the presence of one large cluster of eigenvalues PSSYEVX on 4 nodes is as fast as SSYEVX or even slower, so parallelization does not pay here. The fastest routine here is the divide and conquer routine SSYEVD, but only problems up to a size of $n = 1800$ can be solved hereby. For larger problems the parallel routine PSSYEV is the best performing one. Thus, in the case of one large cluster of eigenvalues PSSYEV is the routine to be used for problem sizes of $n > 1800$ if orthogonality of eigenvectors is strictly necessary.

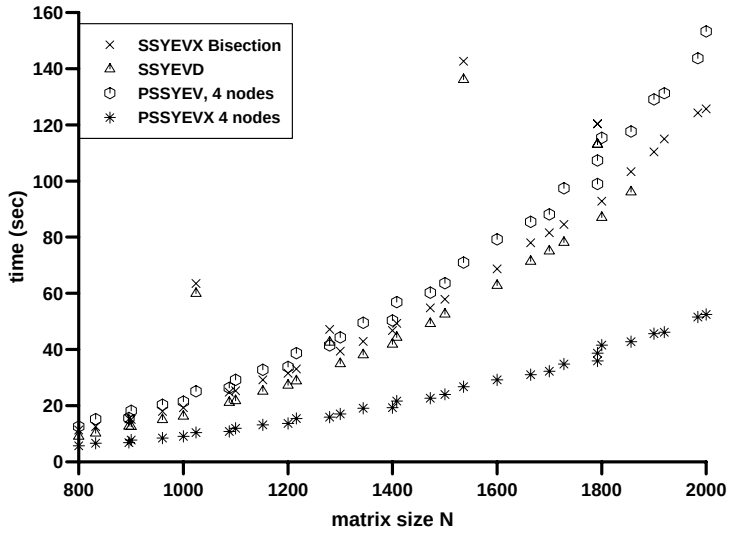

Figure 10: Comparison of the execution times for the best sequential codes with *ScaLAPACK* codes on 4 nodes, computation of all eigenvalues and eigenvectors on CRAY T3E: no large cluster of eigenvalues

7 Performance on CRAY T90

Whereas on CRAY T3E user codes run in dedicated mode, CRAY T90 processors are used within a multiprogramming environment. As a result, on CRAY T3E the user has to pay for the connection time, which is approximately equal to the CPU-time, whereas on CRAY T90 the CPU-time influences the costs, since with increasing problem size CPU-time and real time differ more and more due to other users on the processor(s). For the largest matrix sizes measured, CPU-time and wall clock time differed by a factor of 2 - 3.5.

Some of the more important BLAS in *libsci* on CRAY T90 are parallelized. Whenever the environment variable $NCPUS is set to a larger value than one (the default value for CRAY T90 is 4) and a user or programs in third party libraries like

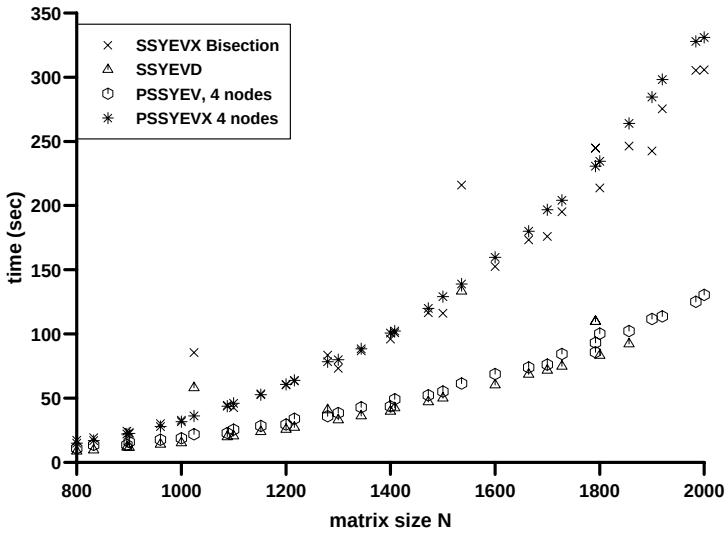

Figure 11: Comparison of the execution times for the best sequential codes with *ScaLAPACK* codes on 4 nodes, computation of all eigenvalues and eigenvectors on CRAY T3E: one large cluster of eigenvalues

NAG is calling one of these *libsci* routines, parallelism is introduced automatically. Former investigations on parallelism in *libsci* for PVP systems are described in [16].

In the following, results are shown for the CPU-time and MFLOPS on one processor, the average number of concurrent CPUs for the usage of two and four processors respectively, the speedup, and the CPU time overhead. The CPU time overhead, i.e. the additional CPU time used in comparison to the sequential version, is mostly influenced by the fact, that on CRAY T90 the attachment of multiple processors is done dynamically. For the calculation of the speedup values the measurements of the wall clock time was used.

To show the effects of multiprogramming, most of the measurements were done in a non-dedicated environment, although the results are not reproducible to some grade. Several measurements of the same problem at different times have shown that the CPU-time and the MFLOPS don't differ significantly whereas the average number of concurrent CPUs is highly influenced by the current load of the machine. In the case of no large clusters of eigenvalues, also results for dedicated calculations are shown.
Some figures contain the results for `SSYEV` and `F02FAE`, which have the same algorithm, to show the influence of multiprogramming.

7.1 No Large Clusters of Eigenvalues

The CPU-times on a single CPU for problems with no large clusters of eigenvalues

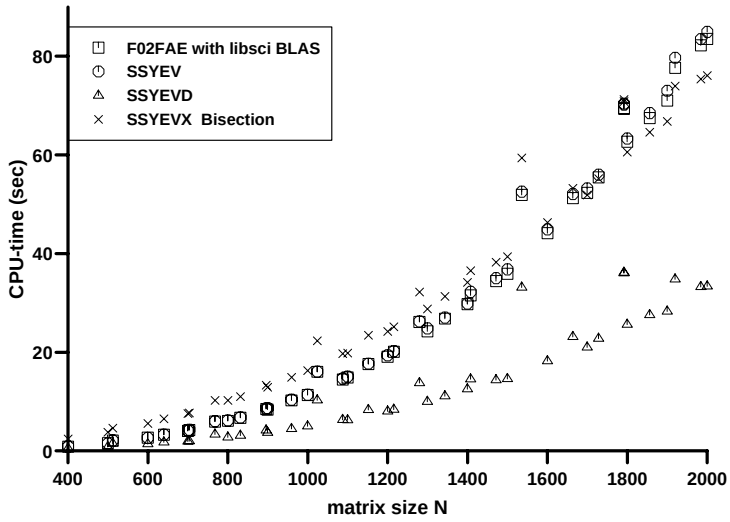

Figure 12: CPU times for different codes, computation of all eigenvalues and eigenvectors on CRAY T90, one CPU: no large clusters of eigenvalues

can be found in Figure 12. The differences between the results for `F02FAE` and `SSYEV` are introduced by multiprogramming. `SSYEVD` shows the most exact eigenvalues but needs the lowest execution time.
On CRAY T3E (see Figure 8) the execution times of `SSYEVX` (bisection) and `SSYEVD` are comparable, whereas on CRAY T90 (see Figure 12) this routine needs about the same time as `SSYEV`.

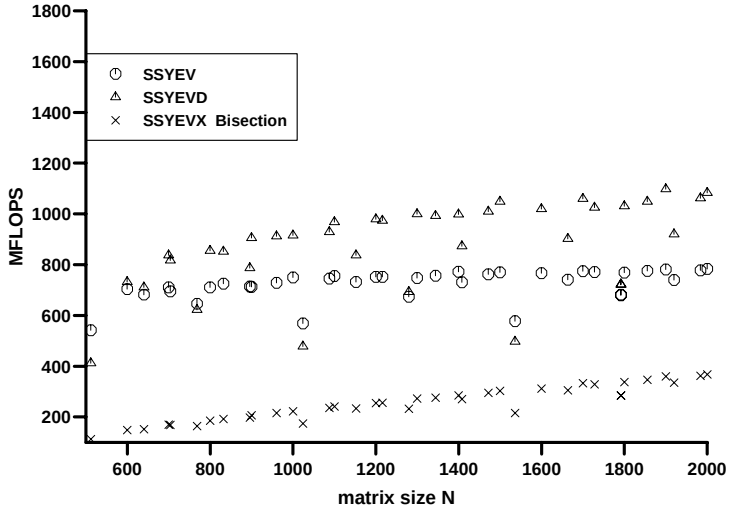

Figure 13: MFLOPS for different codes, computation of all eigenvalues and eigenvectors on CRAY T90, one CPU: no large clusters of eigenvalues

The MFLOPS for problems with no large clusters of eigenvalues are shown in Figure 13. For matrix sizes less or equal 2000, SSYEVD reaches very good values for MFLOPS, up to 61 % of the peak performance, SSYEV up to 44 % but SSYEVX (bisection) only up to 20 %.

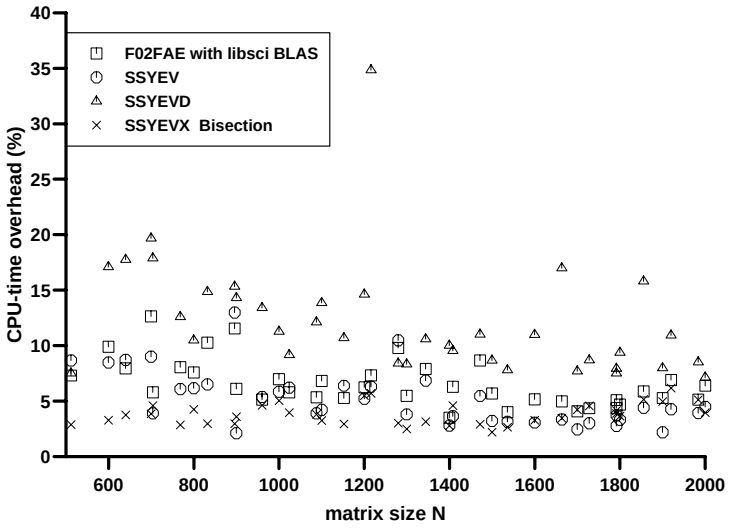

Figure 14: CPU-time overhead using two CPUs for different codes, computation of all eigenvalues and eigenvectors on CRAY T90: no large clusters of eigenvalues

As can be seen from Figure 14, the CPU-time overhead for two CPUs reaches the highest values (about 20 %) for SSYEVD. Tolerable would be a value of about 15 %.

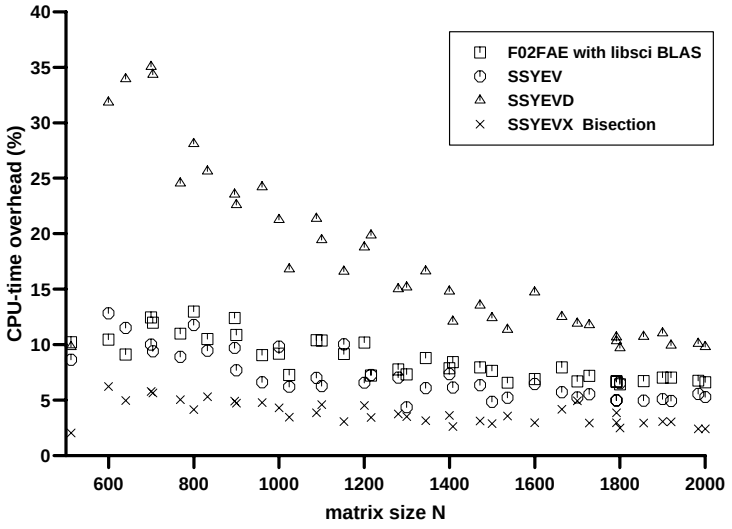

Figure 15: CPU-time overhead using four CPUs for different codes, computation of all eigenvalues and eigenvectors on CRAY T90: no large clusters of eigenvalues

As Figure 15 shows, SSYEVD is the only code showing unacceptable CPU-time overhead using four CPUs: up to 35 % are reached for $n < 1400$.

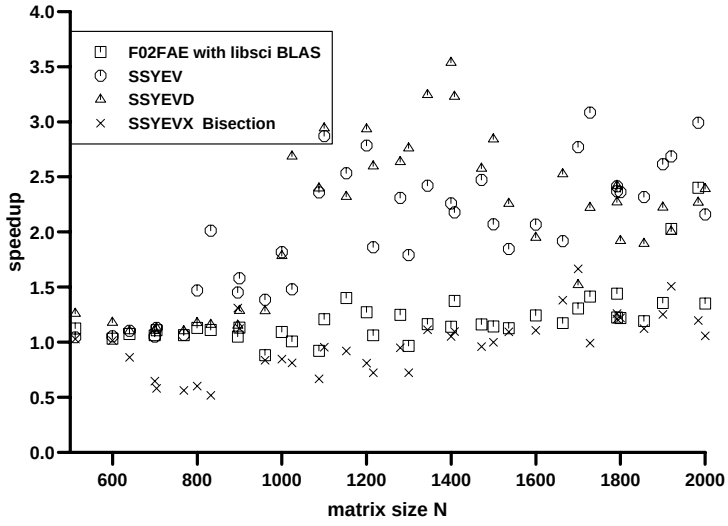

Figure 16: Speedup for two CPUs, computation of all eigenvalues and eigenvectors on CRAY T90: no large clusters of eigenvalues

Figure 16 shows, that for small matrix sizes, where CPU-time and wall-clock time don't differ very much, only very moderate speedup values can be seen. Speedup values greater than two araise because the wall clock time on CRAY T90 is very much influenced by other processes on the processors in a non-dedicated environment.
As can be seen from Figure 17, the code making the best use of two CPUs in a non-dedicated environment is SSYEVD, although the average number of concurrent CPUs decreases for increasing matrix dimension. The values for F02FAE and SSYEV give a hint to the variations which are possible for the same algorithm. SSYEVX (bisection) only reaches values of about 1.1. For this algorithm, a higher grade of parallelization should be possible.
The use of four CPUs (see Figure 18) doesn't make sense for any of the codes checked. Compared to Figure 17, only SSYEVD profits slightly from the two additional CPUs in a non-dedicated environment.

7.1.1 Dedicated Measurements for Matrices With No Large Clusters of Eigenvalues

As Figure 19 compared to Figure 12 shows, the CPU-time and therefore also the MFLOPS don't differ much when measured in a dedicated or non-dedicated environment, respectively.
As can be seen comparing Figure 20 and Figure 14, the CPU-time overhead is also apparent for dedicated measurements.

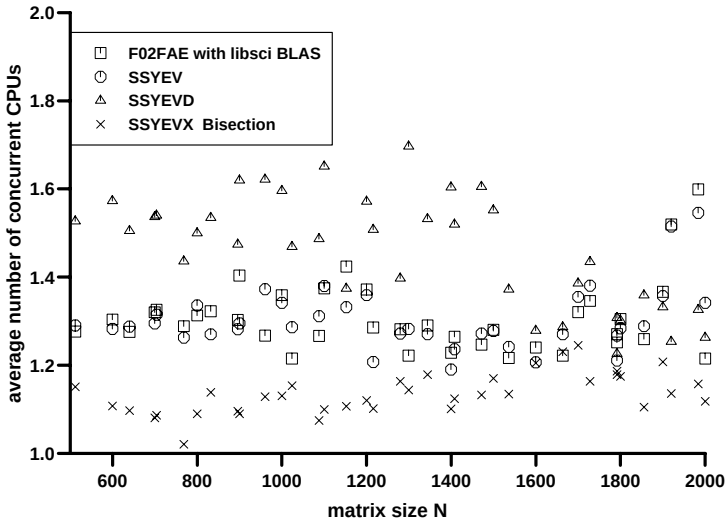

Figure 17: Average number of concurrent CPUs, computation of all eigenvalues and eigenvectors on CRAY T90, two CPUs: no large clusters of eigenvalues

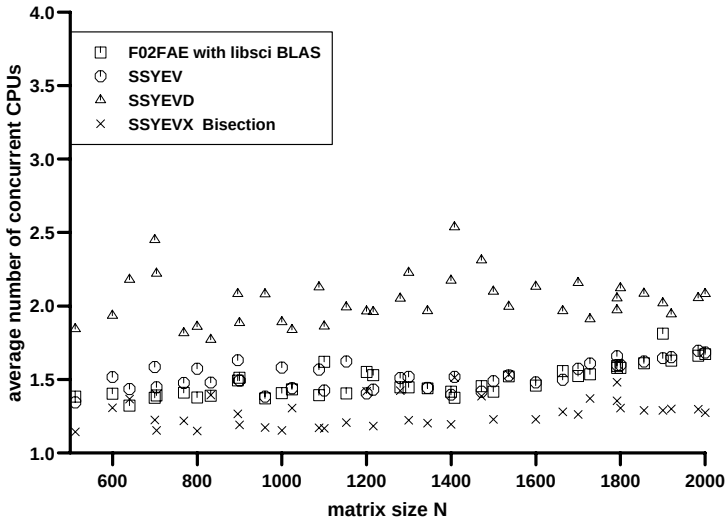

Figure 18: Average number of concurrent CPUs, computation of all eigenvalues and eigenvectors on CRAY T90, four CPUs: no large clusters of eigenvalues

Since CPU-time and real time differ by a factor of up to 3.5 especially for large n for non-dedicated measurements, the speedup values in Figure 16 and 21 cannot be really compared. The speedup values for two CPUs on a dedicated machine show a high grade of parallelization for SSYEV and sufficient results for SSYEVD. The values for SSYEVX are very poor and indicate an inefficient implementation.
Figures 21 and 22 show, that for SSYEV speedup values and the average number of

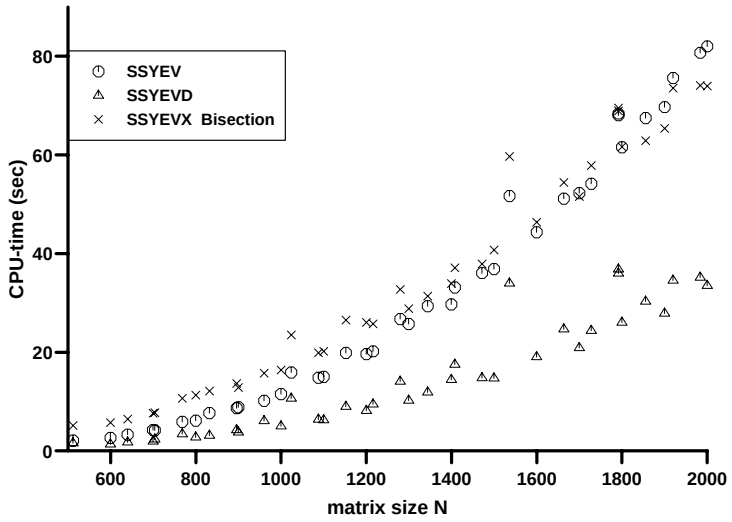

Figure 19: CPU times for different codes, dedicated machine, computation of all eigenvalues and eigenvectors on CRAY T90, one CPU: no large clusters of eigenvalues

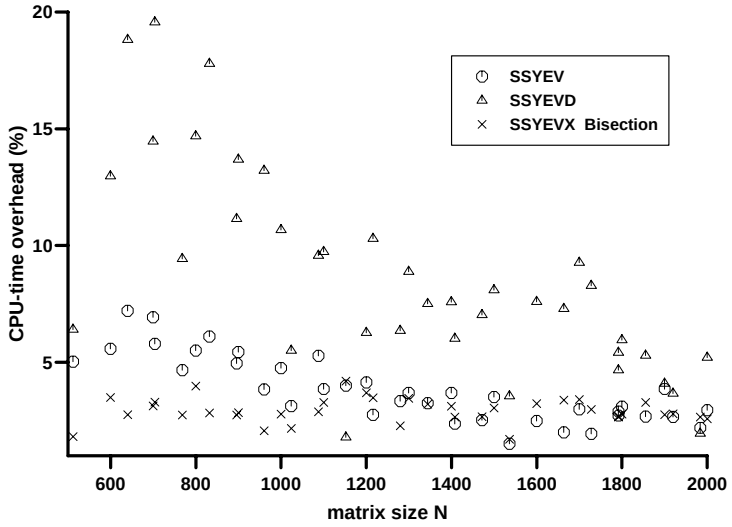

Figure 20: CPU-time overhead using two CPUs, dedicated machine, computation of all eigenvalues and eigenvectors on CRAY T90: no large clusters of eigenvalues

concurrent CPUs are comparable. SSYEVD has an average number of concurrent CPUs of about 1.75, but speedup values of only about 1.6 indicating that perhaps CPUs are kept for the job even during short sequential sections.

The comparison of Figure 17 and 22 shows how much other users prevent the access of several CPUs especially for large matrix dimensions. SSYEV used appro-

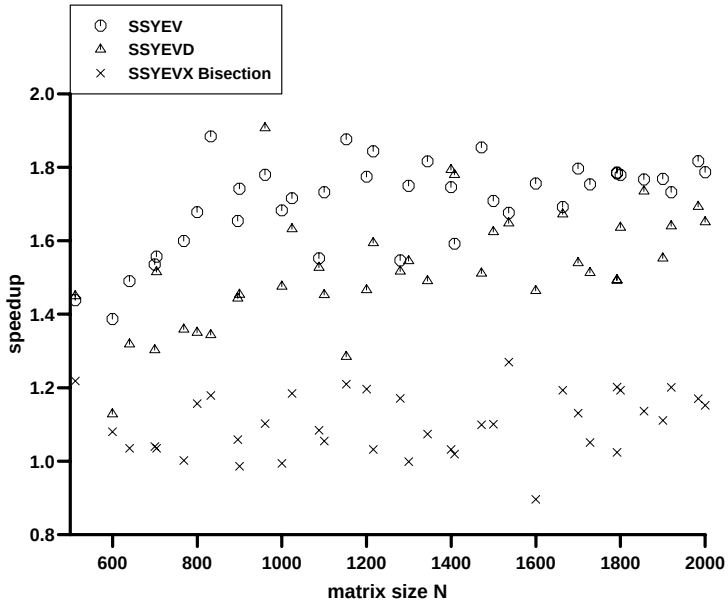

Figure 21: Speedup for two CPUs, dedicated machine, computation of all eigenvalues and eigenvectors on CRAY T90: no large clusters of eigenvalues

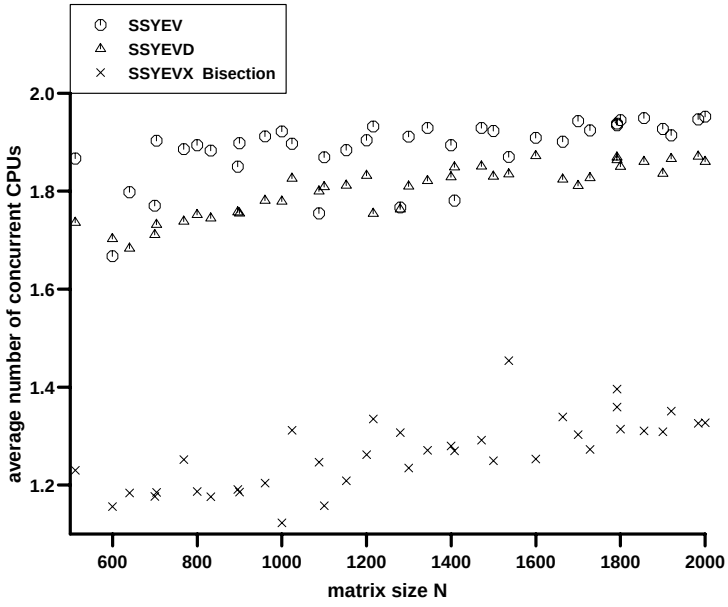

Figure 22: Average number of concurrent CPUs, computation of all eigenvalues and eigenvectors on CRAY T90, two CPUs: no large clusters of eigenvalues, dedicated machine

mately 1.9 CPUs on a dedicated machine, whereas in an non-dedicated environment only about 1.4 CPUs could be used. For SSYEVD these values are 1.75 and 1.4 respectively.

7.2 One Large Cluster of Eigenvalues

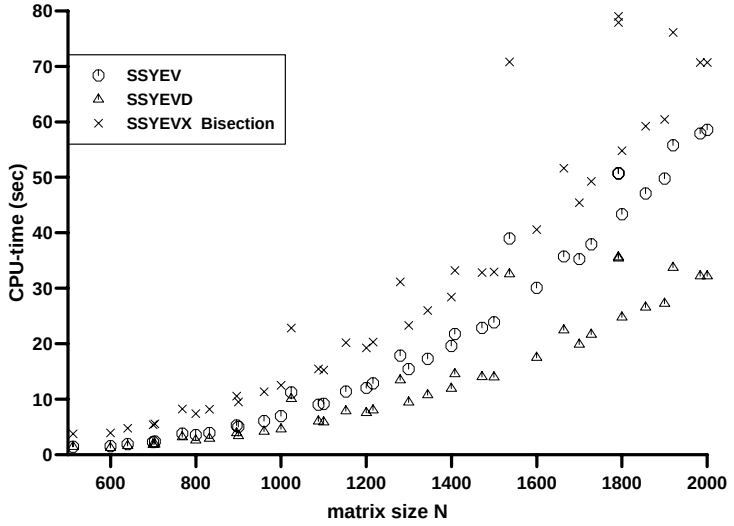

Figure 23: CPU-time for different codes, computation of all eigenvalues and eigenvectors on CRAY T90, one CPU: one large cluster of eigenvalues

Comparing Figure 23 to Figure 12 shows that generally less CPU-time on one CPU is necessary for the computation if there is one large cluster of eigenvalues. Whereas in case of no large clusters of eigenvalues, SSYEVX needs less CPU-time than SSYEV for $n > 1700$, for one large cluster of eigenvalues SSYEVX generally needs more CPU-time than SSYEV. Best performance again shows SSYEVD. On CRAY T3E (see Figure 9), SSYEVX (bisection) needs about the same exeution time as SSYEV, whereas on CRAY T90 (see Figure 23) SSYEVX (bisection) needs more time than SSYEV.

The MFLOPS for problems with one large cluster of eigenvalues are shown in Figure 24. For matrix sizes less or equal 2000, SSYEVD reaches very good values for MFLOPS, up to 59 % of the peak performance, SSYEV up to 47 % and, in contrast to the problems with no large clusters of eigenvalues, SSYEVX (bisection) reaches values up to 40 % instead of 20 %.

As can be seen from Figure 25, all investigated codes show acceptable values for the CPU-time overhead using two CPUs for problems with one large cluster of eigenvalues.

In contrast to the results for no large clusters of eigenvalues (see Figure 15), for small matrix sizes Figure 26 shows unacceptable values for the CPU-time overhead using four CPUs for SSYEVD and SSYEV, too.

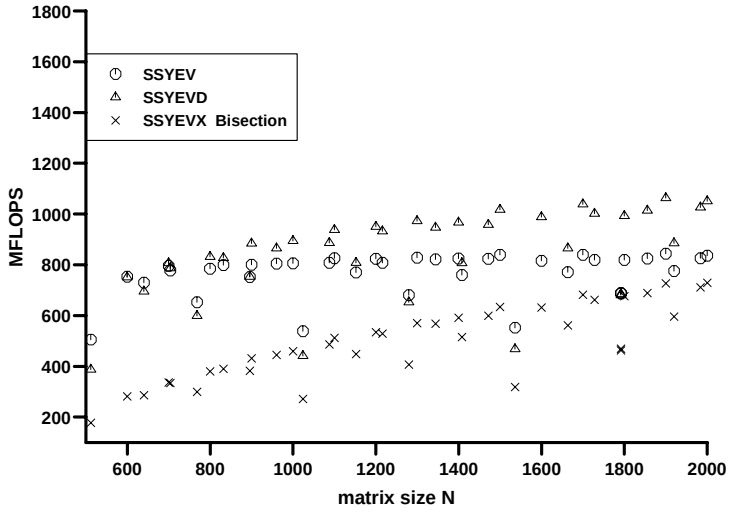

Figure 24: MFLOPS for different codes, computation of all eigenvalues and eigenvectors on CRAY T90, one CPU: one large cluster of eigenvalues

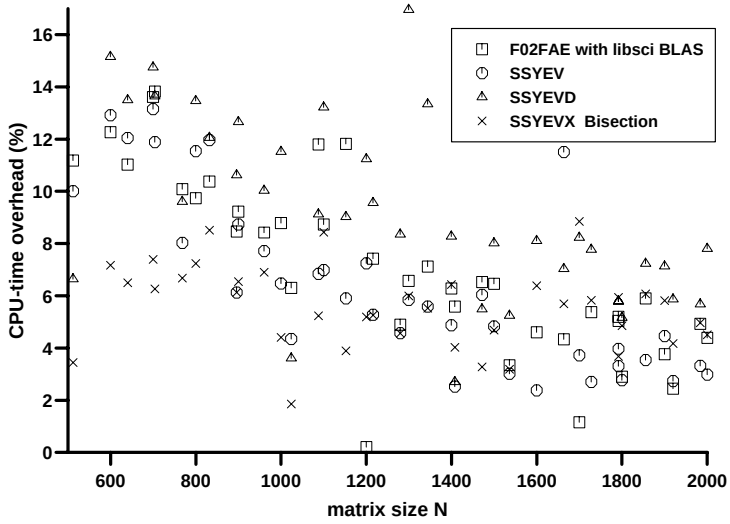

Figure 25: CPU-time overhead using two CPUs for different codes, computation of all eigenvalues and eigenvectors on CRAY T90: one large cluster of eigenvalues

The average number of concurrent CPUs in Figure 27 don't differ much from those in Figure 17. The main difference is, that the values for SSYEV and equivalent codes show a decay over the matrix dimension whereas for no large clusters of eigenvalues the values can be approximated by a constant.
Since mainly less than two CPUs could be attached, the use of four CPUs (see Figure 28) isn't appropriate for any of the tested codes in a non-dedicated environment, especially when the user has to pay for the CPU-time overhead.

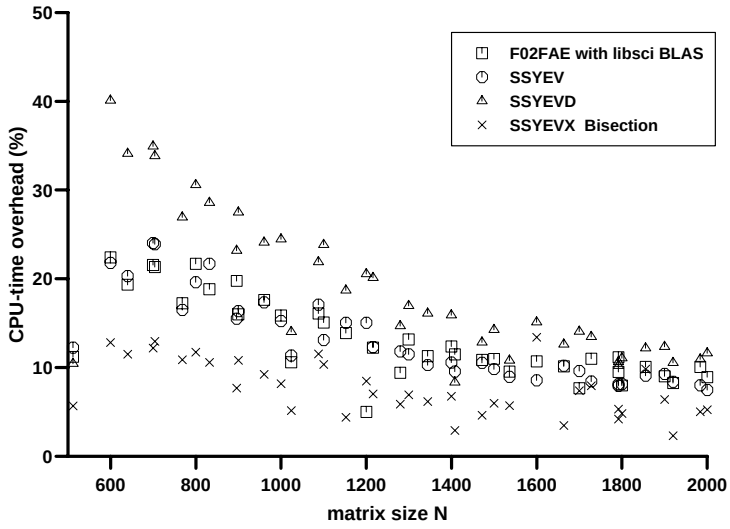

Figure 26: CPU-time overhead using four CPUs for different codes, computation of all eigenvalues and eigenvectors on CRAY T90: one large cluster of eigenvalues

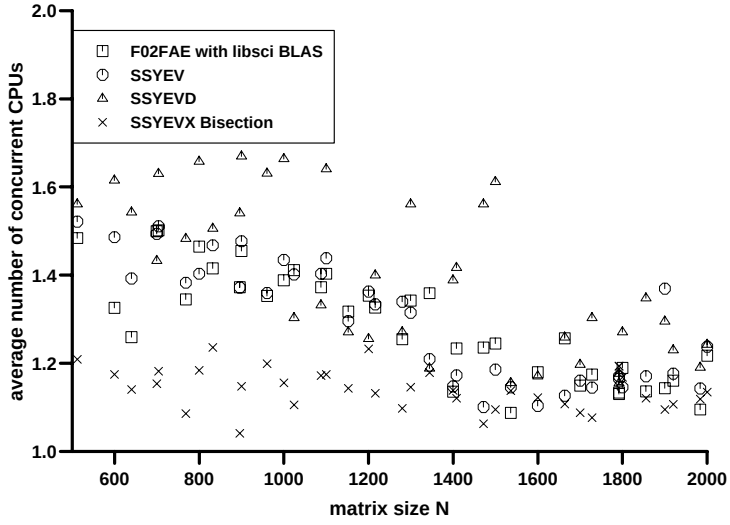

Figure 27: Average number of concurrent CPUs, computation of all eigenvalues and eigenvectors on CRAY T90, two CPUs: one large cluster of eigenvalues

8 Conclusions and Outlook

ScaLAPACK offers very good performance at the expense of a somewhat complicated user interface. Programmers willing to apply *ScaLAPACK* routines should become familiar with the data distribution used in *ScaLAPACK* and adapt their program to this distribution from the start. This will result in good performance

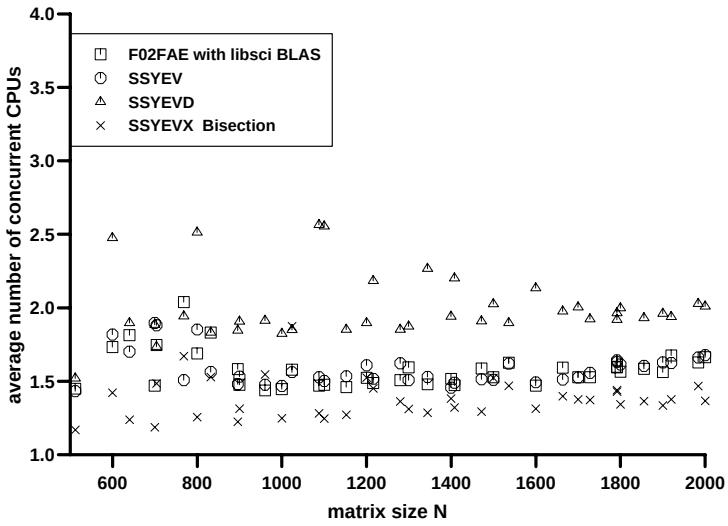

Figure 28: Average number of concurrent CPUs, computation of all eigenvalues and eigenvectors on CRAY T90, four CPUs: one large cluster of eigenvalues

and low memory usage.
Global Arrays mainly offers an infrastructure to treat global objects transparently. If routines from other libraries are to be used performance is lost due to the redistribution of data. There is only one interface routine to *ScaLAPACK* within *Global Arrays*, the one for LU decomposition and the solution of the resulting triangular system. This routine must be modified if other *ScaLAPACK* routines like PSSYEVX shall be used.
The Jacobi solver contained in *NAG Parallel Library* is not suited for a general eigenproblem. There are special cases where it may be better than the general purpose routines of *ScaLAPACK*. *NAG Parallel Library* contains routines for input and output of matrices distributed in the block cyclic distribution necessary for *ScaLAPACK*. This can make use of *ScaLAPACK* easier.

Subroutine libraries like the ones presented here can help developers of new application programs or application packages to take advantage of work already done. With its extensive documentation *NAG Parallel Library* makes parallel programming relatively easy to unexperienced users. There is not only a tutorial for beginners delivered with the library but also one complete example for each user-callable routine which can be modified to fit the needs.
ScaLAPACK also delivers example programs for many of their routines but they are only suited for small applications and as a starting point. Due to the restriction to Fortran 77 there can't be an example which dynamically allocates only the space needed to store the local part of the matrix. The user has to read the Users' Guide where he can find formula how to calculate an upper bound for the local dimensions which he can use with Fortran 77. There the user can also find how to

compute local from global indices and vice versa without the *TOOLS* subroutines. In the examples the global matrices are filled elementwise with calls to PSELSET from *TOOLS* and this is too slow in a large application. So if a potential user of *ScaLAPACK* has access to *NAG Parallel Library* this can make progamming much easier.
The documentation of *Global Arrays 2.4* was not very good; it seemed to be in a preliminary status. Nevertheless the global access to distributed data made it relatively easy to get used to it. With the new version (see below) a better documentation and more examples are delivered, so it will now be even more comfortable to use *Global Arrays*.
Concerning the PVP routines, SSYEVD is the best choice. It is the fastest sequential algorithm and additionally parallelizes well. On CRAY T3E, SSYEVD on one node is as fast as the best parallelized code PSSYEV on 4 nodes for the matrices with one large cluster of eigenvalues. The implementation of SSYEVX (bisection) on CRAY T90 seems to be ineffective. It shows a bad grade of parallelization and also the performance isn't as good as on CRAY T3E.

Compared to the message-passing programming model the usage of the PVP system is much easier as there is no need of distributing data and calling routines with new parameters. There is no difference in the usage of CRAY T90 and a workstation except setting the $NCPUS environment variable.
On both machines turnaround time is influenced by other users, on CRAY T90 because of the multiprogramming environment, on CRAY T3E because the user has to wait until the number of nodes he needs are free. So there is no chance to compare throughput on both machines.
Comparing CPU-times needed it can be seen that on CRAY T3E there is always one parallel program that needs about the same or less CPU time on 16 nodes of CRAY T3E than on one node of CRAY T90.

From the measurements in the dedicated environment it can be seen that most PVP routines investigated have high potential for shared-memory parallelism. Thus on clusters of multiprocessor nodes with shared memory where whole nodes can be used in a dedicated manner there is a good chance of getting high speedups on a single node with shared-memory parallelism and additional speedup with coarse-grain parallelism on a higher level using message-passing.

In the next release of *ScaLAPACK* there will be the new reduction algorithm and a parallel version of the divide and conquer algorithm used in SSYEVD. As this was the best performing sequential algorithm this sounds promising. It should be well parallelizable, too.

LAPACK also has announced a new release 3.0 with a new eigensolver.

Since November 1999 *Global Arrays 3.0* is available with a new and much better

User's Manual. It allows global arrays of up to seven dimensions and contains an additional package for distributed I/O: Disk Resident Arrays. In *NWChem* now *PeIGS 3* is integrated but the eigensolver still is based on BLAS 1 calls. We did not investigate those new versions.

References

[1] L.S. Blackford, J. Choi, A.Cleary et al.
ScaLAPACK Users' Guide
SIAM Philadelphia, 1997
Also available via WWW under
http://www.netlib.org/scalapack/slug/scalapack_slug.html

[2] Silicon Graphics Computer Systems / Cray Research
Scientific Libraries Reference Manual, Volume 1-2,
SR-2081 3.0

[3] Global Arrays User Guide
Only available via WWW under
http://www.emsl.pnl.gov:2080/docs/global/ga.html

[4] NAG Parallel Library Manual
Release 2
The Numerical Algorithms Group Limited, 1997

[5] NAG Fortran Library - Mark 17
The Numerical Algorithms Group Limited, 1995

[6] D. Elwood, G. Fann, R. Littlefield
Parallel Eigensystem Solver PeIGS Version 2.1, rev. 0.0
Pacific Northwest Laboratory July 28, 1995
Only available via WWW under
http://www.emsl.pnl.gov:2080/docs/nwchem/doc/peigs/docs/peigs3.html

[7] Dynamic Memory Allocator Homepage
http://www.emsl.pnl.gov:2080/docs/parsoft/ma/MAapi.html

[8] K.S. Stanley
Execution Time of Symmetric Eigensolvers
Dissertation, University of California at Berkeley, 1997
Available via WWW under
http://www.cs.berkeley.edu/~stanley

[9] E. Anderson, Z. Bai, C. Bischof et al.
LAPACK Users' Guide, Second Edition,
SIAM Philadelphia, 1995

Also available via WWW under
http://www.netlib.org/lapack/lug/lapack_lug.html

[10] J. Demmel, K. Stanley
The Performance of Finding Eigenvalues and Eigenvectors of Dense Symmetric Matrices on Distributed Memory Computers
LAPACK Working Note 86
In Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing. SIAM 1994
Also available via WWW under
http://www.netlib.org/lapack/lawns/

[11] G. Henry
Improving Data Re-Use in Eigenvalue-Related Computations,
PhD thesis, Cornell University,
Ithaca, NY, January 1994

[12] J. Galarowicz, B. Mohr
Analyzing Message Passing Programs on the CRAY T3E with PAT and VAMPIR
Forschungszentrum Jülich GmbH, Zentralinstitut für Angewandte Mathematik, Interner Bericht FZJ-ZAM-IB-9809, Mai 1998,
man pat on Cray systems

[13] Cray Research
Introducing the MPP Apprentice Tool, Cray Manual IN-2511, 1994

[14] Guide to Parallel Vector Applications, Cray Manual 004-2182-002, January 1999

[15] PCL - The Performance Counter Library
http://www.fz-juelich.de/zam/PCL/

[16] P. Jansen, M. Marx, W.E. Nagel, M. Romberg, M. Vaeßen, and R. Zimmermann
Cray Scientific Library (libsci) and Parallelism
-Some Performance Aspects-
Forschungszentrum Jülich GmbH, Zentralinstitut für Angewandte Mathematik, Interner Bericht KFA-ZAM-IB-9310, August 1994

[17] T. Oechtering
Performance Measurements of BLACS Routines on CRAY T3E
Forschungszentrum Jülich GmbH, Zentralinstitut für Angewandte Mathematik, Interner Bericht FZJ-ZAM-IB-2000-06, July 2000

A Tables

Table 6: Percentage of time spent in communication routines: no large clusters of eigenvalues

percentage of time spent in comm/wait		small problem	large problem
PSSYEVX	4 nodes	28-32 %	6 - 8 %
	8 nodes	37-41 %	12-14 %
	16 nodes	$\approx$ 45 %	12-14 %
	25 nodes	$\approx$ 48 %	$\approx$ 18 %
	32 nodes	$\approx$ 46 %	17-20 %
	36 nodes	50-51 %	$\approx$ 19 %
	64 nodes	52-55 %	$\approx$ 18 %
PSSYEV	4 nodes	19-23 %	2 - 3 %
	8 nodes	27-31 %	3 - 5 %
	16 nodes	30-31 %	4 - 6 %
	25 nodes	29-31 %	5 - 9 %
	32 nodes	30-33 %	6 -11 %
	36 nodes	34-36 %	$\approx$ 9 %
	64 nodes	$\approx$ 33 %	$\approx$ 11 %
GA_DIAG_STD	4 nodes	34 %	14-16 %
	8 nodes	34 %	18-20 %
	16 nodes	37 %	20-21 %
	25 nodes	44 %	21-23 %
	32 nodes	41 %	21-23 %
	36 nodes	42 %	24 %
	64 nodes	44 %	25 %
F02FQFP	4 nodes	13 %	8 %
	8 nodes	16 %	8 %
	16 nodes	23 %	13 %
	25 nodes	14 %	10 %
	32 nodes	21 %	14 %
	36 nodes	22 %	13 %
	64 nodes	24 %	not measured

The range of values of the *ScaLAPACK* routines for small problems was taken from the different block sizes and, for $np = 8$ or $np = 32$, also different grid shapes. For the large problems we often only show the value for $nb = 20$ as with $nb = 32$ we ran into the performance problem of `SGEMM` and communication costs raise to twice or even three times as much as with $nb = 20$ as the nodes with local matrices not equal 1024×1024 have to wait for those with 1024×1024 local parts (see section 4.4). For the smaller numbers of nodes values given in the tables are for n larger than or equal to the matrices given in Table 1 whereas for $np \geq 25$ the values are only for the size given in Table 1.
All values given in Tables 8 and 9 are the values for the grid shape and block size which gave the shortest execution time for the given matrix size.
The operation counts for *ScaLAPACK* and *Global Arrays* are taken from *PCL* and

Table 7: Percentage of time spent in communication routines: one large cluster of eigenvalues

percentage of time spent in comm/wait		small problem	large problem
PSSYEVX	4 nodes	43 %	61 %
	8 nodes	56 %	too large
	16 nodes	77 %	too large
	25 nodes	85 %	too large
	32 nodes	88 %	too large
	36 nodes	89-90 %	too large
	64 nodes	94 %	too large
PSSYEV	4 nodes	30-31 %	1 - 3 %
	8 nodes	$\approx$ 32 %	3 - 5 %
	16 nodes	33-35 %	$\approx$ 7 %
	25 nodes	31-35 %	$\approx$ 6 %
	32 nodes	33-35 %	7 -12 %
	36 nodes	34-44 %	$\approx$ 10 %
	64 nodes	35-39 %	$\approx$ 12 %
GA_DIAG_STD	4 nodes	33 %	$\leq$ 20 %
	8 nodes	39 %	$\leq$ 20 %
	16 nodes	36 %	$\leq$ 20 %
	25 nodes	39 %	20 %
	32 nodes	41 %	21 %
	36 nodes	40 %	20 %
	64 nodes	50 %	22 %
F02FQFP	4 nodes	20 %	4 %
	8 nodes	11 %	6 %
	16 nodes	13 %	9 %
	25 nodes	17 %	12 %
	32 nodes	14 %	13 %
	36 nodes	16 %	11 %
	64 nodes	19 %	not measured

rounded to three digits. For *NAG* the operation counts were determined by *PAT*. The MFLOPS rates were computed from these operation counts and the measured times.

The values for `F02FQFP` were measured for the matrix dimensions given in Table 1. It can be seen from Table 8, that for no large clusters of eigenvalues the number of floating point operations is very much higher than for the other codes. Generally, `F02FQFP` needs less operations and lower execution time for the problems with one large cluster of eigenvalues (Table 9), but the values for the large problem show a strange behaviour: if the number of columns on each processor is even, (that is the case for the measurements for 4 nodes up to 32 nodes) the number of operations, compared to the values for no large clusters of eigenvalues, is reduced by a factor of up to 3.5, whereas for 36 nodes (number of columns 167 and 155 on the last processor) the factor is only 1.2.

Table 8: Millions of floating-point operations and MFLOPS: no large clusters of eigenvalues

Mill. operations per node (MFLOPS per node)		small problem	large problem
PSSYEVX	4 nodes	82-92 (64-72)	8190-8380 (176-182)
	8 nodes	87-94 (56-61)	10800-11600 (165-178)
	16 nodes	144-170 (53-63)	16100-17500 (157-170)
	25 nodes	177-205 (49-57)	19400-20200 (174-181)
	32 nodes	219-245 (50-61)	22000-22700 (176-182)
	36 nodes	203-248 (45-55)	23200-24100 (174-181)
	64 nodes	268-331 (46-56)	30700-32100 (175-183)
PSSYEV	4 nodes	175-184 (76-80)	16800-17000 (128-129)
	8 nodes	192-198 (67-69)	25700-26600 (138-142)
	16 nodes	340-405 (58-69)	37100-38600 (131-137)
	25 nodes	458-491 (61-66)	47000-47800 (134-136)
	32 nodes	455-631 (49-68)	50400-54200 (124-133)
	36 nodes	486-642 (49-65)	55700-59300 (123-131)
	64 nodes	674-945 (46-65)	74600-81100 (116-127)
GA_DIAG_STD	4 nodes	124-186 (69-103)	14500-22000 (77-117)
	8 nodes	126-201 (57-91)	19300-31600 (64-105)
	16 nodes	236-389 (51-85)	27100-46000 (56-96)
	25 nodes	293-488 (47-79)	33600-57700 (52-90)
	32 nodes	349-585 (45-76)	37600-64800 (46-80)
	36 nodes	360-566 (43-68)	40700-69400 (48-82)
	64 nodes	496-768 (39-61)	54500-92900 (44-75)
F02FQFP	4 nodes	750-778 (57-61)	101000-105000 (52-54)
	8 nodes	802-853 (46-49)	135000-143000 (54-57)
	16 nodes	1410-1540 (51-55)	187000-202000 (49-53)
	25 nodes	2120-2310 (52-57)	255000-280000 (52-57)
	32 nodes	2430-2640 (43-47)	292000-317000 (44-48)
	36 nodes	807-2690 (14-48)	334000-357000 (51-55)
	64 nodes	3220-3570 (44-47)	not measured

The MFLOPS cannot be very high, since only BLAS 1 routines are used in `F02FQFP`. The values are nearly the same for all measurements in Table 8 and for the small problems in Table 9. For the large problems in Table 9 the MFLOPS are lower for an even number of matrix columns per processor.

Table 9: Millions of floating-point operations and MFLOPS: one large cluster of eigenvalues

Mill. operations per node (MFLOPS per node)		small problem	large problem
PSSYEVX	4 nodes	74-157 (49-104)	7840-53900 (24-163)
	8 nodes	79-337 (24-107)	too large
	16 nodes	134-1930 (10-149)	too large
	25 nodes	166-4340 (6-156)	too large
	32 nodes	212-7140 (5-155)	too large
	36 nodes	194-8250 (4-157)	too large
	64 nodes	260-21900 (2-157)	too large
PSSYEV	4 nodes	149-157 (73-77)	16800-17000 (147-149)
	8 nodes	164-171 (64-67)	23400-23600 (140-145)
	16 nodes	295-346 (57-66)	32700-33900 (136-141)
	25 nodes	396-428 (59-63)	41700-42400 (137-140)
	32 nodes	406-552 (48-66)	45100-48300 (127-136)
	36 nodes	431-555 (49-63)	49800-52700 (127-134)
	64 nodes	582-813 (45-63)	60100-70000 (108-126)
GA_DIAG_STD	4 nodes	138-228 (60-105)	15600-31900 (60-120)
	8 nodes	138-258 (45-85)	19500-31800 (45-75)
	16 nodes	257-530 (40-85)	28000-65400 (40-95)
	25 nodes	329-694 (35-80)	34500-82300 (35-85)
	32 nodes	390-843 (35-75)	38000-92000 (30-80)
	36 nodes	408-817 (35-70)	41600-99000 (30-80)
	64 nodes	496-1070 (30-65)	54200-132000 (30-70)
F02FQFP	4 nodes	841-849 (51-52)	5200-57900 (32-36)
	8 nodes	793-857 (42-46)	55000-69400 (26-33)
	16 nodes	1340-1500 (45-50)	62600-80900 (20-25)
	25 nodes	1710-2010 (40-47)	53300-73000 (15-21)
	32 nodes	1990-2240 (38-43)	62600-88100 (14-20)
	36 nodes	681-2190 (14-45)	247000-294000 (40-48)
	64 nodes	3260-3910 (41-49)	not measured

List of Figures

List of Tables